AL-TR-1992-0056

ARMSTRONG LABORATORY

INFORMATION INTEGRATION FOR CONCURRENT ENGINEERING (IICE) IDEF4 OBJECT-ORIENTED DESIGN METHOD REPORT

Richard J. Mayer, PhD
Arthur Keen
M. Sue Wells

KNOWLEDGE BASED SYSTEMS, *INCORPORATED*
2726 LONGMIRE
COLLEGE STATION, TEXAS 77845

HUMAN RESOURCES DIRECTORATE
LOGISTICS RESEARCH DIVISION

MAY 1992

INTERIM TECHNICAL REPORT FOR PERIOD FEBRUARY 1991 - APRIL 1992

AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6573

NOTICES

[illegible]

MICHAEL K. PAINTER
Program Manager

[illegible]

BERTRAM W. CREAM, Chief
Logistics Research Division

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE May 1992	3. REPORT TYPE AND DATES COVERED Interim - February 1991 to April 1992

4. TITLE AND SUBTITLE IDEF4 Object-Oriented Design Method Manual	5. FUNDING NUMBERS C F33615-90-C-0012 PE 63106F PR 2940 TA 01 WU 08
6. AUTHOR(S) Richard J. Mayer, PhD M. Sue Wells Arthur A. Keen	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Knowledge Based Systems, Inc. 2726 Longmire College Station, Texas 77845	8. PERFORMING ORGANIZATION REPORT NUMBER KBSI-IICE-90-STR-01-0592-01
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Armstrong Laboratory Human Resources Directorate Logistics Research Division Wright-Patterson AFB, OH 45433	10. SPONSORING/MONITORING AGENCY REPORT NUMBER AL-TR-1992-0056
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	12b. DISTRIBUTION CODE A

13. ABSTRACT *(Maximum 200 words)*

This document provides a method overview, practice and use description, and language reference for the IDEF4 Object-Oriented Design Method. The name IDEF originates from the Air Force program for Integrated Computer-Aided Manufacturing (ICAM) from which the first ICAM Definition, or IDEF, methods emerged. It was in recognition of this foundational work, and in support of an overall strategy to provide a family of mutually-supportive methods for enterprise integration, that continued development of IDEF technology was undertaken. More recently, with their expanded focus and widespread use as part of Concurrent Engineering, Total Quality Management (TQM), and business re-engineering initiatives, the IDEF acronym has been re-cast as the name referring to an integrated family of Integration Definition methods. IDEF4 was developed as a design method to assist in the production of quality designs for object-oriented implementations. This document is targeted at both object-oriented programmers looking for a design method and programmers learning an object-oriented programming language who want to know how to design good object-oriented programs.

14. SUBJECT TERMS object-oriented, IDEF, programming, method, methodology, information systems, software design, systems engineering	15. NUMBER OF PAGES 137 16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std Z39-18
298-102

Contents

Figures

Foreword

The Department of Defense (DoD) has long recognized the opportunity for significant technological, economic, and strategic benefits attainable through the effective capture, control, and management of information and knowledge resources. Like manpower, materials, and machines, information and knowledge assets are recognized as vital resources that can be leveraged to achieve competitive advantage. The Air Force Information Integration for Concurrent Engineering (IICE) program, sponsored by the Armstrong Laboratory's Logistic Research Division, was established as part of a commitment to further the development of technologies that will enable full exploitation of these resources.

The IICE program was chartered with developing the theoretical foundations, methods, and tools to successfully implement and evolve towards an information-integrated enterprise. These technologies are designed to leverage information and knowledge resources as the *key* enablers for high quality systems that achieve better performance in terms of both life-cycle cost and efficiency. The subject of this report is one of a family of methods that collectively constitute a technology for leveraging available information and knowledge assets. The name IDEF originates from the Air Force program for Integrated Computer-Aided Manufacturing (ICAM) from which the first ICAM Definition, or IDEF, methods emerged. It was in recognition of this foundational work, and in support of an overall strategy to provide a family of mutually-supportive methods for enterprise integration, that continued development of IDEF technology was undertaken. More recently, with their expanded focus and widespread use as part of Concurrent Engineering, Total Quality Management (TQM), and business re-engineering initiatives, the IDEF acronym has been re-cast as the name referring to an integrated family of Integration Definition methods. Before discussing the development strategy for providing an integrated family of IDEF methods, however, the following paragraphs will briefly introduce what constitutes a method.

Method Anatomy

A **method** is an organized, single-purpose discipline or practice (Coleman, 1989). A method may have a formal theoretic foundation. However, most do not (except possibly in the eyes of the developer of the method). Generally, methods evolve as a distillation of *best-practice* experience in a particular domain of cognitive or physical activity. The term **methodology** has at least two common usages. The first use is to refer to a class of similar methods. So, one may hear reference to the *function modeling methodology* referring to methods such as IDEFØ[1] and LDFD.[2] In an other sense, the term **methodology** is used to refer to *a collection of methods and tools, the use of which is governed by a process superimposed on the whole* (Coleman, 1989). Thus, it is common to hear the criticism that a tool (or method) has no underlying methodology. Such a criticism is often leveled at a tool (or method) which has a graphical language but for which no procedure for the appropriate application of the language or use of the resulting models is provided. For simplicity, the term **tool** is used to refer to a software system designed to support the application of a method.

Though a method may be thought of informally as simply a procedure for performing a task plus perhaps a representational notation, it may be described more formally as consisting of three components as illustrated in Figure F-1. Each method has (a) a definition, (b) a discipline, and (c) many uses. The definition specifies the basic intuitions and motivation behind the method, the concepts involved, and the theory of its operation. The discipline includes the procedure by which the method is applied and the language, or syntax, of the method. The procedure associated with the method discipline provides the practitioner with a reliable process for achieving consistently good results. The method syntax is provided to eliminate ambiguity among those involved in the development of complex engineering products. Many system analysis and engineering methods

[1]ICAM Definition method for Function Modeling
[2]Logical Data Flow Diagramming method

use a graphical syntax to provide visualization of collected data in such a way that key information can be easily extracted.[3] The third element of the

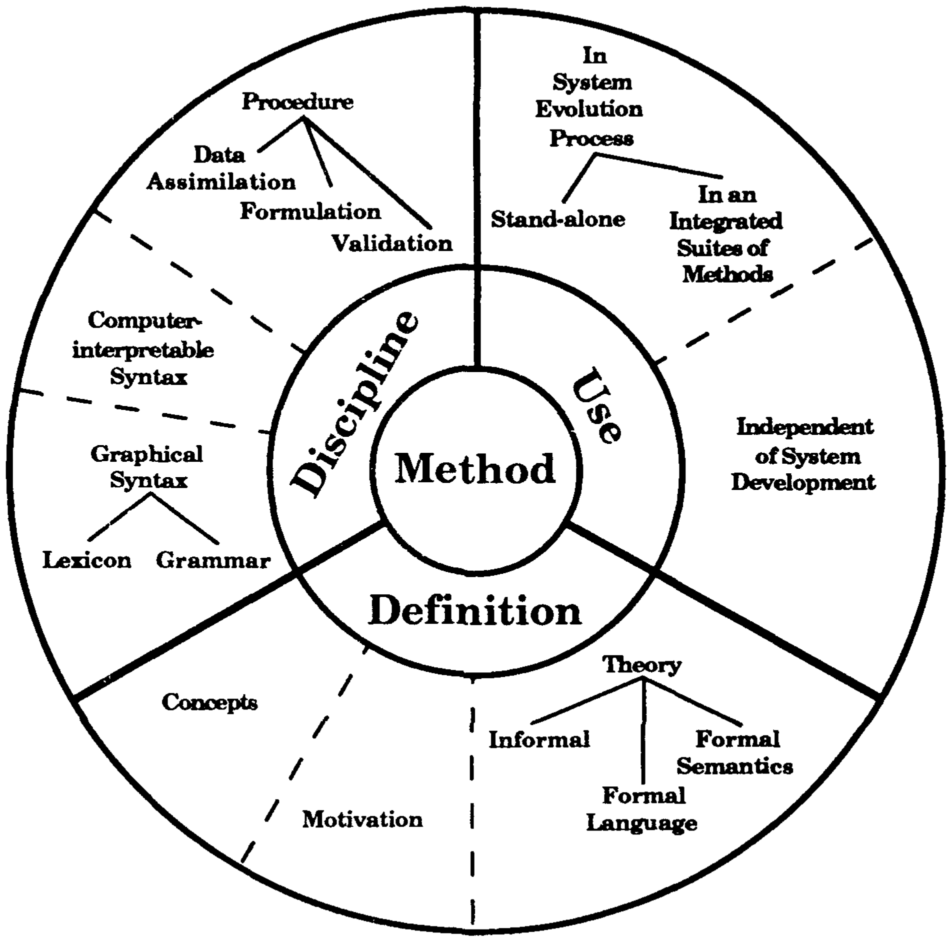


Figure F-1
Anatomy of a Method

method anatomy, the use component, focuses on the context-specific application of the method.

---

[3]Graphical facilities provided by a method language serve not only to document the analysis or design process undertaken, but more importantly, to highlight important decisions or relationships that must be considered during method application. The uniformities to which an expert becomes attuned over many years of experience are thus formally encoded in visualizations that emulate expert sensitivities.

Ultimately, methods are designed to facilitate a scientific approach to problem solving. This goal is accomplished by first, helping one understand the important objects, relations, and constraints that must be discovered, considered, or decided on; and second, by guiding the method practitioner through a disciplined approach, consistent with good-practice experience, towards the desired result. Formal methods, then, are specifically designed to raise the performance level (quality and productivity) of the novice practitioner to something comparable with that of an expert (Mayer, 1987).

Family of Methods

As Mr. John Zachman, in his seminal work on information systems architecture observed, "...there is not <u>an</u> architecture, but a <u>set</u> of architectural representations. One is not right and another wrong. The architectures are different. They are additive, complementary. There are reasons for electing to expend the resources for developing each architectural representation. And, there are risks associated with <u>not</u> developing any one of the architectural representations." Consistent, reliable creation of correct architectural representations, whether they be artificial approximations of a system (models) or purely descriptive representations, requires the use of a guiding method. These observations underscore the need for <u>many</u> "architectural representations," and correspondingly many methods.

Methods, and their associated architectural representations, focus on a limited set of system characteristics and explicitly ignore those that are not directly pertinent to the task at hand. Methods were never intended to evaluate and represent every possible state or behavioral characteristic of the system under study. If such a goal were achievable, the exercise would itself constitute building the actual system, thus negating the benefits to be gained through method application (e.g., problem simplification, low cost, rapid evaluation of anticipated performance, etc.).

The search for a single method, or modeling language, to represent all relevant system life cycle and behavioral characteristics, therefore, would necessitate skipping the design process altogether. Similarly, the search for a single method to facilitate conceptualization, system analysis, and design continues to frustrate those making the attempt.

Recognizably, the plethora of special-purpose methods which typically provide few, if any, explicit mechanisms for integration with other methods, is equally frustrating. The IDEF family of methods is intended to strike a favorable balance between special-purpose methods whose effective application is limited to specific problem types, and "super methods" which attempt to include all that could ever be needed. This balance is maintained within the IDEF family of methods by providing explicit mechanisms for integrating the results of individual method application.

Critical method needs identified through previous studies and research and development activities[4] have given rise to renewed effort in IDEF method integration and development activities, with an explicit mandate for compatibility among the family of IDEF methods. Providing for known method needs with a family of IDEF methods was not, however, the principal goal of methods engineering activity within the IICE program. The primary emphasis for these efforts was directed towards establishing the foundations for an engineering discipline guiding the appropriate selection, use, extension, and creation of methods that support integrated systems development in a cost-effective and reliable manner.

New methods development has struck out where known and obvious method voids existed (rather than re-inventing existing, and often very good

[4]Of particular note is the Knowledge-Based Integrated Information Systems Engineering (KBIISE) Project conducted at the Massachusetts Institute of Technology (MIT) in 1987 where a collection of highly qualified experts from academic and research organizations, government agencies, computer companies, and other corporations identified method and tool needs for large-scale, heterogeneous, distributed systems integration. See Defense Technical Information Center (DTIC) reports A195851 and A195857.

methods) with the explicit mission to forge integration links with and between existing IDEF methods. When applied in a stand-alone fashion, IDEF methods serve to embody knowledge of good practice for the targeted fact collection, analysis, design, or fabrication activity. As with any good method, the IDEF methods are designed to raise the performance level of novice practitioners to a level that is comparable to that of an expert by focusing attention on important decisions while masking out irrelevant information and unneeded complexity. Viewed collectively as a complementary toolbox of methods technology, the IDEF family is designed to promote integration of effort in an environment where global competitiveness has become increasingly dependent upon the effective capture, management, and use of enterprise information and knowledge assets.

Preface

This document provides a method overview, practice and use description, and language reference for the IDEF4 Object-oriented Design Method developed under the Information Integration for Concurrent Engineering (IICE) project, F33615-90-C-0012, funded by Armstrong Laboratory, Logistics Research Division, Wright-Patterson Air Force Base, Ohio 45433, under the technical direction of United States Air Force Capt. Michael K. Painter. The prime contractor for IICE is Knowledge Based Systems, Inc. (KBSI), College Station, Texas. Dr. Paula S. deWitte is IICE Project Manager at KBSI. Dr. Richard J. Mayer is Principal Investigator, and Mr. Arthur A. Keen is Methods Engineering Thrust Manager.

KBSI acknowledges the technical input (Knowledge Based Systems Laboratory, 1991) to this document made by previous work under the Integrated Information Systems Evolutionary Environment (IISEE) project associated with the Knowledge Based Systems Laboratory, Department of Industrial Engineering, Texas A&M University.

Acronyms

CDS	Contract Data Sheets........8
CIDS	Class-invariant Data Sheets........5
CLOS	Common Lisp Object System........2
FORTRAN	**For**mula **Tran**slator........13
ID	Identifier........46
IICE	Information Integration for Concurrent Engineering..... xii
KBSI	Knowledge Based Systems, Inc. xii
LISP	**Lis**t **P**rocessing........22
ODBMS	Object Data Base Management System........65
OOD	Object-oriented Design 2
OOPL	Object-oriented Programming Language xi
OSTN	Object-state Transition Network........112
UOB	Unit of Behavior 111

1.0 Executive Summary

The purpose of this document is to provide a comprehensive description of the IDEF4 Object-oriented Design method. Its intention is to guide a person in becoming proficient in the application of IDEF4 to produce quality object-oriented designs.

The scope of this document includes 1) a description of the IDEF4 syntax and the motivation for including each syntax element, 2) a description of the development procedure for an object-oriented design using IDEF4, and 3) example applications of IDEF4. Within this scope, the IDEF4 Method Report is designed for the following audience:

1. object-oriented programmers looking for a design language, and
2. programmers learning an object-oriented programming language (OOPL) who want to know how to design good object-oriented programs.

The motivation for developing the IDEF4 method was the potential misuse of the powerful object-oriented paradigm which could result in poor quality designs. The object-oriented philosophy and development practice have demonstrated the ability to produce code that exhibits desirable life-cycle qualities such as modularity, maintainability, and reusability. In addition, the object-oriented programming paradigm has demonstrated major advancements in the ease with which software code can be created, enabling people to produce and maintain code efficiently. Paradoxically, the ease with which this type of code can be produced also makes it easier to create software of poor design. This results in systems that are not modular, are difficult to maintain, and whose implementations are far more difficult to reuse. The goal of IDEF4 is to assist in the correct application of the object-oriented programming technology to ensure more effective use of that technology.

2.0 Introduction

It is widely recognized that the modularity, maintainability, and code reusability resulting from the object-oriented programming paradigm can be realized in traditional data processing applications. The proven ability of this paradigm to support data-level integration in large, complex, distributed systems is also a major factor in the widespread interest from the traditional data processing community. The object-oriented approach provides the developer with an abstract view of the program as composed of a set of state-maintaining objects that define the behavior of the program by the protocol of their interactions.

Languages for object-oriented programming include the Common Lisp Object System (CLOS) (Keene, 1989), C++ (UNIX System, 1989), Smalltalk (Goldberg, 1978), Object Pascal (Macintosh Programmer's Workshop, 1989), and others. IDEF4 is a design method for software designers who use such object-oriented languages. Since effective use of the object-oriented paradigm requires a thought process different from that used with conventional procedural or database languages, standard methodologies such as structure charts, data flow diagrams, and traditional data design models (hierarchical, relational, and network) are inappropriate. The IDEF4 Object-oriented Design (OOD) Method seeks to provide the necessary facilities to support the object-oriented design decision-making process. The primary design goals of IDEF4 are:

1. to provide support for creating object-oriented designs whose implementations will exhibit desirable life-cycle qualities and reduce total implementation development time, and

2. to make it easy to evaluate object-oriented code to determine both conformation to design and desired life-cycle qualities.

IDEF4 maintains information about an object-oriented design in a manner that will preserve many of the conceptual and notational advances made in previous method-development efforts. Such consistency with previous work should help practicing software engineers learn and effectively use the IDEF4 modeling techniques. Furthermore, IDEF4 encapsulates the "best practice" experience of designing for object-oriented databases and programming environments. As such, IDEF4 is focused on the identification, manipulation, display, and analysis of the following.

1. Object definitions - including attributes and their types.
2. Object structures - including the inheritance hierarchy or lattice and individual object composition relative to other objects.
3. Individual object behavior - including method specifications and constraints (pre-, post-, and during conditions) governing instantiation, deletion, and other behaviors required by the object.
4. Protocol of the system routines - including location of object routines in the object inheritance lattice. The protocols also include:
    - the type, number, and ordering of the arguments;
    - an abstract description of the behavior of each protocol item; and
    - the specialization of abstract behavior for individual object types.

As a design method, IDEF4 is structured to specify the components of an object-oriented system design that must be managed during the design phase of a system development process. It employs a unique organizational mechanism to ensure that design models do not become cumbersome and difficult to use with increasingly larger projects.

Conceptually, an IDEF4 design model consists of two submodels: the class submodel and the method submodel (see Figure 2-1). The two submodels are linked through a dispatch mapping. These two structures capture all the information represented in a design model. The submodels are a very broad grouping of a particular type of information. A data sheet displays very localized, often textual information, related to a particular data type or method set. Except for the protocol and client diagrams, which are usually small, the diagrams indicated in Figure 2-1 range in size from single boxes to representations of all the information of a particular kind that is available from an entire submodel. Diagrams are views of the submodels. The dispatch mapping connects the two submodels and is expressed by certain optional annotations on the inheritance diagrams and on the method taxonomy diagrams. In Figure 2-1, the sharp-cornered boxes represent the submodel types and dispatch mapping; the round-cornered boxes represent types of diagrams and data sheets of an IDEF4 model.

In other words, IDEF4 divides the OOD activity into discrete, manageable chunks. Each subactivity is supported by a graphical syntax that highlights the design decisions that must be made and their impact on other perspectives of the design. No single diagram shows all the information contained in the IDEF4 design model, thus limiting confusion and allowing rapid inspection of the desired information. Carefully designed overlap among diagram types serves to ensure compatibility between the different submodels.

Due to the size of the class and method submodels, the designer never sees these structures in their entirety. Instead, the designer makes use of the collection of smaller diagrams that effectively capture the information represented in the class and method submodels. The diagram types that make up the class submodel are 1) inheritance diagrams, 2) protocol diagrams, 3) type diagrams, and 4) instantiation diagrams. The diagram types that make up the method submodel are 1) method taxonomy diagrams and 2) client diagrams. These six diagram types are used in an IDEF4 design to express 1) the compositional structure of the data object classes using type diagrams, 2) inheritance relations using inheritance diagrams, 3) method contracts using method taxonomy diagrams, 4) protocols of the object-oriented design using protocol diagrams, and 5) functional decomposition using client diagrams. The following sections contain an explanation of the diagrams and a brief description of their purpose.

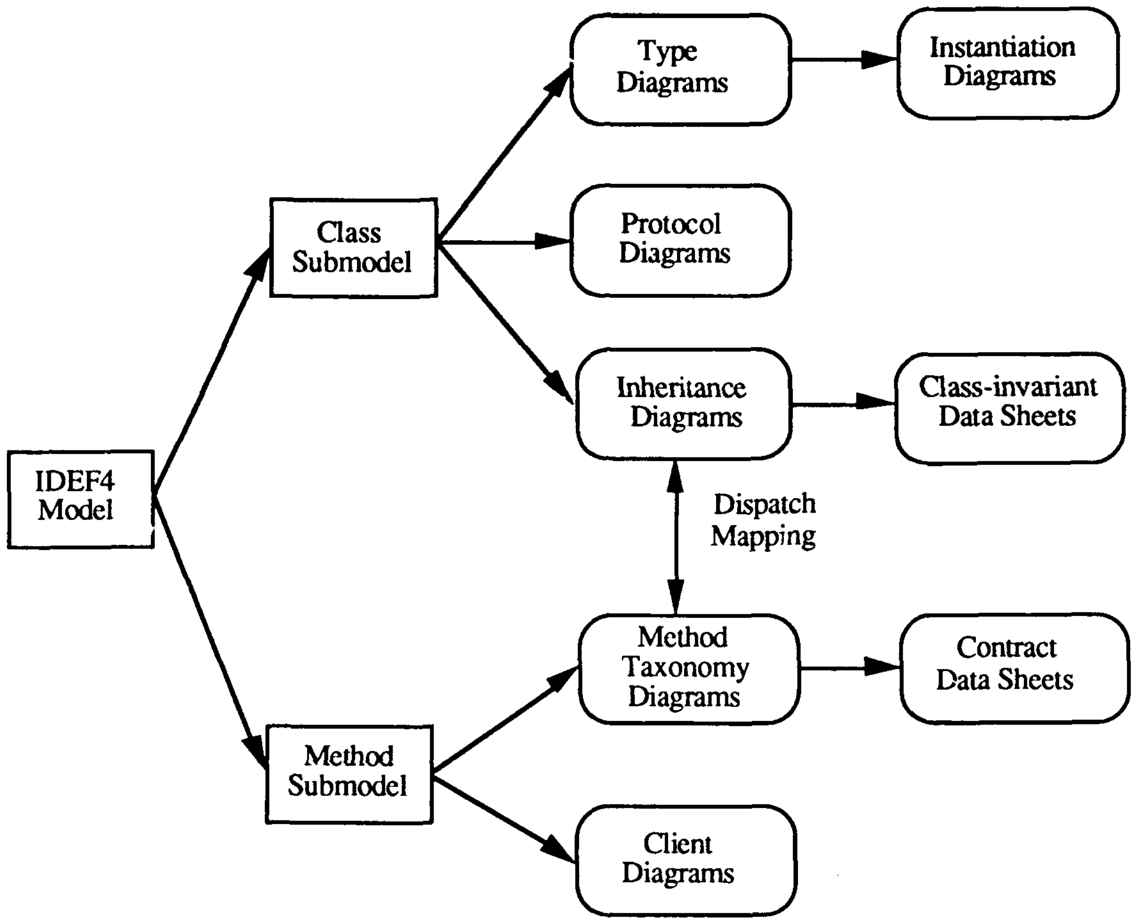
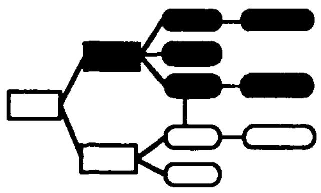


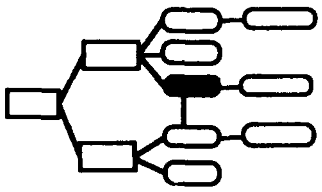
Figure 2-1
Organization of the IDEF4 Model

2.1 Class Submodel



The class submodel consists of protocol diagrams, type diagrams, inheritance diagrams, instantiation diagrams, and class-invariant data sheets (CIDS). The class submodel shows class-inheritance and class composition structure.

2.1.1 Inheritance Diagrams



Inheritance diagrams specify the inheritance relations among classes. For example, Figure 2-2 shows the class *Filled-rectangle* inheriting structure and behavior directly from the classes *Rectangle* and *Filled-object* and indirectly from the class *Object*.

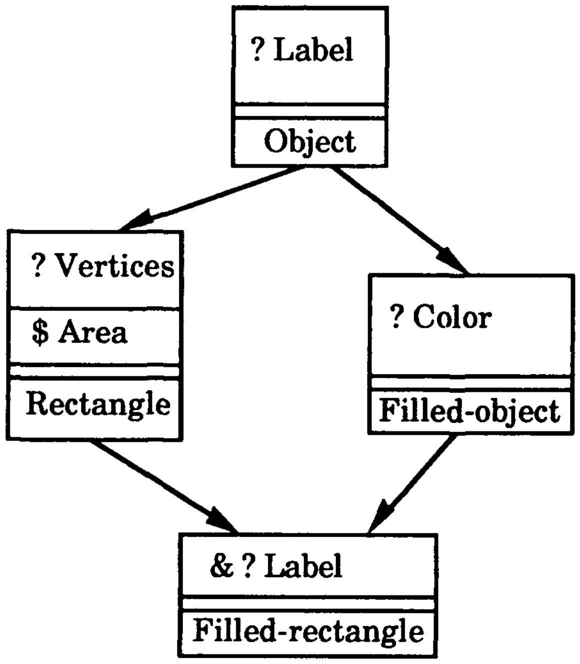
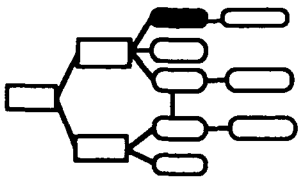


Figure 2-2
Inheritance Diagram

2.1.2 Type Diagrams



Type Diagrams specify relations among classes defined through attributes of one class that has values which are instances of another class, or that are composed of instances of that class. Figure 2-3 shows a type diagram in which the *2-Wheeled* class has features *Wheel-1* and *Wheel-2*. These features are shown to return instances of the class *Wheel* that returns a value of type *Real* for its feature *Diameter*.

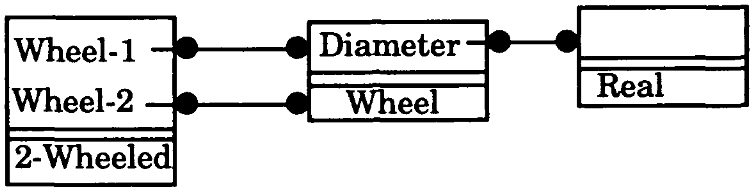
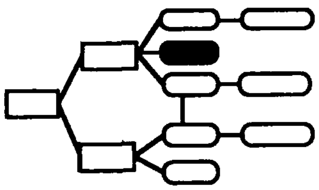


Figure 2-3
Type Diagram

2.1.3 Protocol Diagrams



Protocol Diagrams specify the class argument types for method invocation. Figure 2-4 illustrates a protocol diagram for the *Fill-closed-object: Polygon*

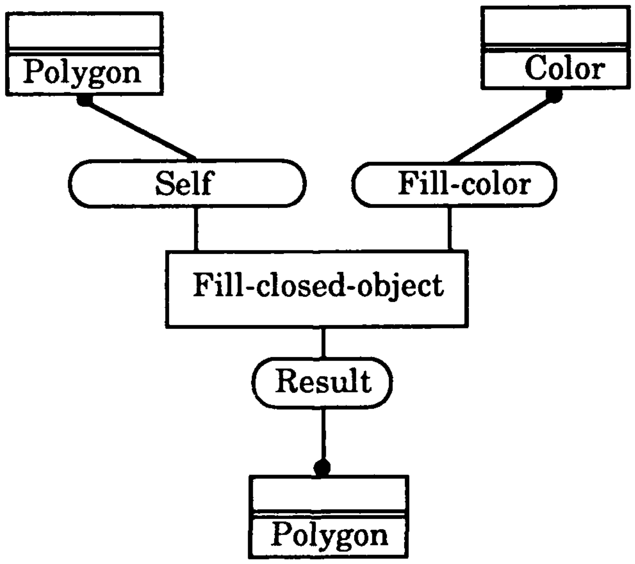


Figure 2-4
Protocol Diagram

routine-class pair. From the diagram, it is apparent that *Fill-closed-object* will accept an instance of the class *Polygon* as its primary (self) argument and an instance of the class *Color* as a secondary argument, and will return an instance of a *Polygon.*

2.1.4 Instantiation Diagrams

Instantiation diagrams are associated with type diagrams in the class submodel. The instantiation diagrams describe the anticipated situations of composite links between instantiated objects that are used to validate the design.

2.2 Method Submodel

The method submodel consists of client diagrams, method taxonomy diagrams, and contract data sheets (CDSs). An overview of each one of these components is given in the following sections.

2.2.1 Method Taxonomy Diagrams

Method taxonomy diagrams classify method types by behavioral similarity. A method taxonomy diagram classifies a specific system behavior type according to the constraints placed on the method sets represented in the taxonomy. The arrows indicate additional constraints placed on the method sets. Figure 2-5 shows a *Print* method taxonomy diagram. The method sets in the taxonomy are grouped according to the additional contracts placed on the methods in each set. In the example, the first method set, *Print*, has a contract which states that the object must be printable. The *Print-text* method-set contract would have constraints such as “the object to be printed must be text.”

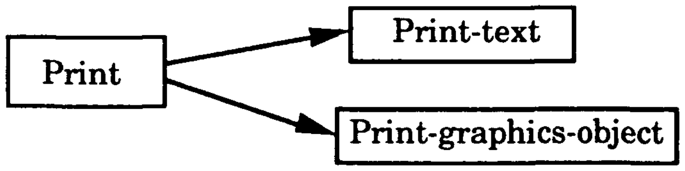
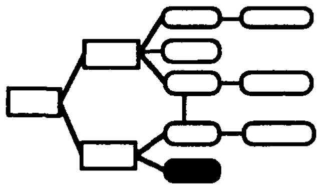


Figure 2-5
Method Taxonomy Diagram

2.2.2 Client Diagrams



Client diagrams illustrate clients and suppliers of routine-class pairs. Double-headed arrows point from the routine that is called to the calling routine. Figure 2-6 shows a client diagram for which the *Redisplay* routine attached to the class *Redisplayable-object* calls the *Erase* routine of the *Erasable-object* class and the *Draw* routine on the *Drawable-object* class.

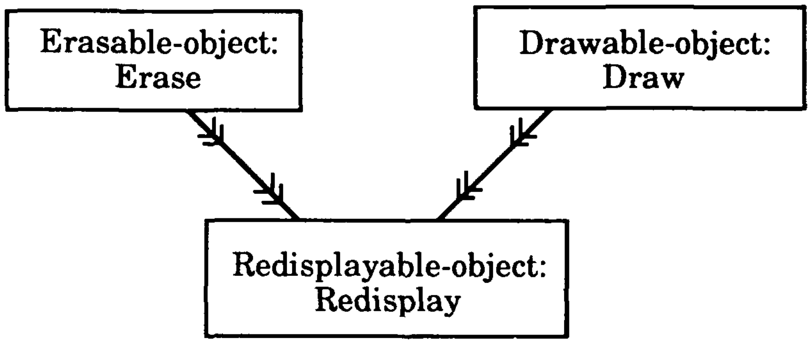
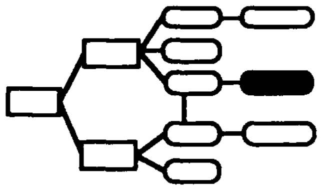


Figure 2-6
Client Diagram

2.3 Data Sheets

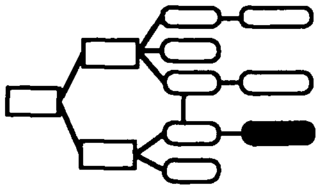
There are also two specialized data sheets that accompany the diagrams, class-invariant data sheets (CIDS) and contract data sheets (CDS). Understanding the information conveyed by these diagrams requires a description of the basic concepts of IDEF4 that is presented in the next chapter of this document.

2.3.1 Class-invariant Data Sheets



Class-invariant data sheets are associated with inheritance diagrams and specify constraints that apply to every instance of a particular object class. There is one CIDS for each class.

2.3.2 Contract Data Sheets



CDSs are associated with the method sets in method taxonomy diagrams and specify contracts that the implemented methods in a method set must satisfy. There is one contract data sheet for each method set.

3.0 IDEF4 Object-oriented Concepts

IDEF4 is a method for object-oriented design; it is not an object-oriented programming language. It was developed to be used by designers of object-oriented systems regardless of the object-oriented language used for implementation of the system. However, discussion of the components of an IDEF4 design must be preceded by an understanding of the basic concepts of object-oriented programming languages. In this section, the basic object-oriented programming elements and their representation in an IDEF4 design will be described. In addition, the basic elements of object-oriented programming (classes, features, and methods) will be introduced along with an elementary description of how they would appear in an IDEF4 model.

The object-based tradition is often identified with message passing (i.e., a message containing the name of an operation and arguments is sent to a receiver). As with the data-driven tradition, the receiver and the name of the operation are jointly used to select a method to invoke (Gabriel, 1991).

There are four focuses for object-oriented systems (see Figure 3-1): object-centric, class-centric, operation-centric, and message-centric. Object-centric systems provide a minimal amount of data abstraction, i.e., "state" and "operations" with no special provision for classes. Class-centric systems give primacy to classes. Classes 1) describe the structure of objects, 2) contain the methods defining the behavior, 3) provide inheritance topologies, and 4) specify data sharing between objects. Operation-centric systems give primacy to operations: the operations themselves contain all the behavior for objects but they may use classes or objects to describe inheritance and sharing. Message-centric systems give primacy to messages: messages are first class, and carry operations and data from object to object. In class-centric and object-centric systems, the class or object is highlighted rather than the operations.

Dimensions	Message-centric	Operation-centric
Class/type-centric	Smalltalk C++ virtuals	CLOS FORTRAN Overloads C++ Overloads
Object-centric	Self Actors	Prolog Data-driven

Figure 3-1
Dimensions of Object-oriented Languages (Gabriel, 1991)

3.1 Classes

Object-oriented programming is a very intuitive style of programming in that programs are made through the creation and manipulation of objects. This style of programming, since it is more natural than traditional programming styles, makes writing code easier for more people. Paradoxically, the ease provided by this orientation also makes it easier for more people to produce code that is inefficient, difficult to maintain, and not reusable. One of the primary goals of IDEF4 is to help take full advantage of the technology and avoid potential pitfalls.

Programs written in an object-oriented language define classes and methods for classes. A novice in object-oriented programming will often confuse the terms type, class, and object.

“Type” has at least five different meanings (Gabriel, 1991) including: declaration types, representational types, signature types, methodical types, and value types. Declaration types are used to specify an invariant for what values may be stored in a particular variable or structure. Value types refer to sets of objects that may be stored in a particular variable or structure. A representational type is one that defines the storage layout of objects (i.e., double in C++). A signature type is defined by the kinds of operations that may be performed on objects. For example, the kinds of operations that may be performed on a number and a vertex are quite different. A methodical type is one for which methods can be written. Classes in most object-oriented languages are examples of methodical types.

A class is a type (methodical type) and an object is an instance of a type, therefore an instance of a class. The description of a class is the definition of a set of local, state-defining attributes and of a set of methods that define the behavior of instances of that class and their relationship to instances of other types that make up the system. In other words, a class is a data structure that includes a set of state-defining attributes and a set of routines that apply to objects of that class. An IDEF4 design will describe classes and methods associated with those classes.

3.1.1 Classes vs. Objects

An important concept common to object-oriented and structured design and programming is encapsulat*ion or information* hiding. Encapsulation is attempting to divide a program into a number of modules to minimize intermodule interaction or coupling (Yourdon & Constantine, 1979). Therefore, before a variable or procedure can be exported from a module, there must be an explicit declaration.

The term *class* is heavily overloaded in the object-oriented approach. It refers to:

- categories, or types, of objects in the real world (real-world perspective);
- data types representing categories of objects (data-item perspective); and
- modules of associated operations that define data types (module perspective).

The term *object* is also overloaded. It refers to:

- real-world objects (real-world perspective); and

- data items belonging to one class or another (data-item perspective).

An object is an instance of a class. It is often convenient to merge the real-world and data-item perspectives of *class*, and the real-world and data-item perspectives of *Object*. This allows us to think of an object as a real-world object that has been tagged with a few extra characteristics needed to keep track of it inside a computer and to think of classes as categories of such objects. This allows us to think of the class as an object that is tagged with the information used to manipulate the computer representations of the instances of the class. In other words, the computer views objects as real-world objects. The ability to represent these concepts easily is precisely the advantage that object-oriented languages have over structured programming languages such as Pascal and C.

The classes in an object-oriented system define the types of objects that exist within the system. Each class contains a set of feature definitions that characterize the state and behavior of the instances of that class. The set of feature definitions consists of attributes and methods. The attribute definitions are used by the instances of the class to store their state. The methods characterize the behavior of instances of the class.

Classes are the major syntactic construct in IDEF4, as in all object-oriented formalisms. In IDEF4, a class is represented by a square-cornered box (see Figure 3-2) with the name of the class listed below the double line at the bottom of the box. IDEF4 requires that the first letter of the class name be capitalized. The features of the class are also displayed in the Class Box with private features (i.e., those intended for use within instances of the class only) displayed below the export line and with public features (i.e., those to be documented, advertised and supported for external use) displayed above the export line. The ability to define the public and private features is supported explicitly by object-oriented languages such as C++, Eiffel (Meyer, 1988), and Flavors. Various feature symbols, prefixed to the feature name, may also be used to provide additional information about the role that the feature plays. For each class defined, IDEF4 allows the attachment of class-invariant constraints using class-invariant data sheets as discussed in Section 3.1.3. These class-invariant constraints represent additional information about a class that is true for all objects in the class at all times. The class-invariants described in a design will provide constraints on the implementation of the design and serve as part of the specifications for a class. All the concepts introduced in this section will be addressed in greater detail later in this document.

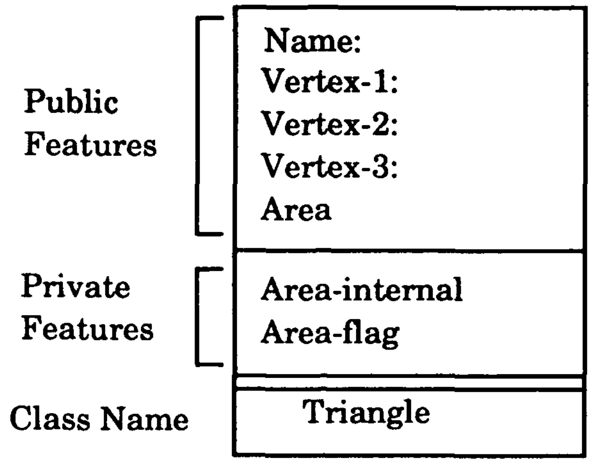
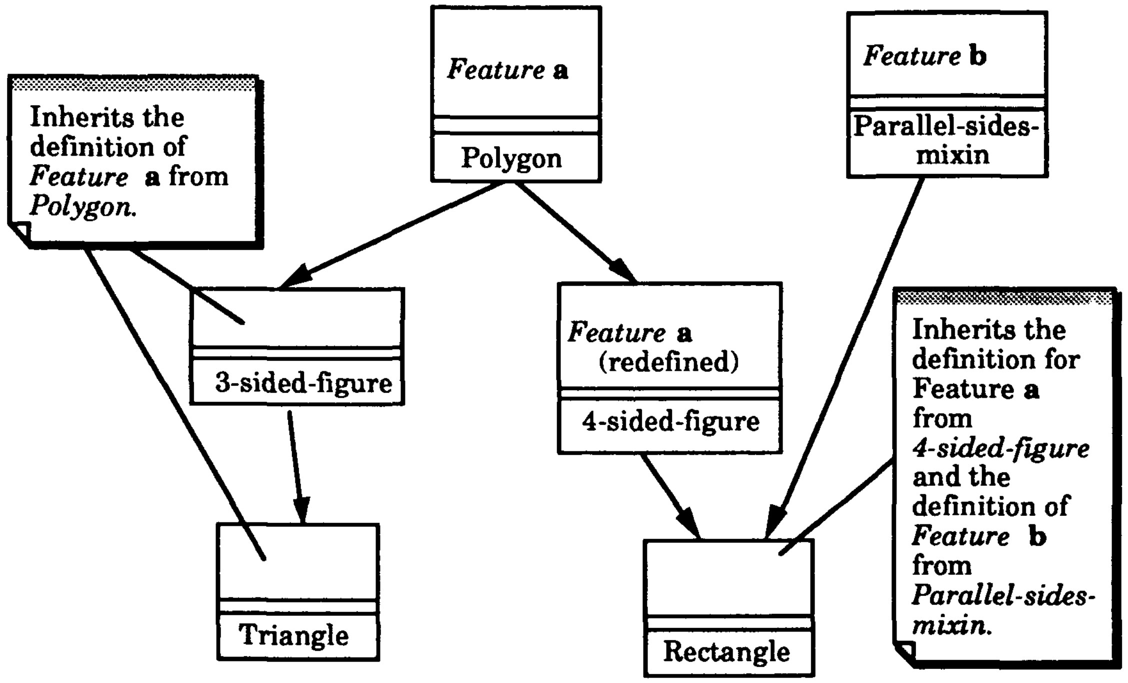


Figure 3-2
Class Box in IDEF4

3.1.2 Class-inheritance

Perhaps the most distinguishing characteristic of an OOPL is inheritance, especially multiple inheritance. Multiple inheritance occurs when a class (a more specific class) inherits features (attributes and methods) from more than one superclass (more general class or classes). The concept of inheritance in OOPL provides a means of organizing instances of classes into related sets and allows for the reuse of methods and class features within the inheritance hierarchy. From the real-world point of view, the inheritance phenomenon operates like a specialization relation. That is, the inheriting class (subclass) is a specialization of the class from which it inherits (superclass).

Figure 3-3 illustrates the approach taken in IDEF4 for modeling a class-inheritance hierarchy. (In the illustration, the details that are irrelevant to this discussion are omitted for clarity.) The arrows, in the diagram, point from superclass to subclass. In this illustration, *3-sided Figure* is a subclass of the class *Polygon*. From this description, it is indicated that any object that is a *3-sided Figure* is also a specialization of a polygon. Furthermore, any behavior exhibited by a polygon will also be exhibited by a *3-sided Figure* unless the behavior is specialized in the definition of the *3-sided Figure* class. In the illustration, objects of type *Rectangle* inherit the characteristics and methods from the *Parallel-sides-mixin* and the *4-sided Figure* classes. In the example, the class *4-sided Figure* is a direct subclass of *Polygon* and the class *Rectangle* is an indirect subclass of class *Polygon*. Each subclass inherits the characteristics and methods associated with its direct and indirect superclass(es).



Feature a
Polygon
Feature b
Parallel-sides-mixin
Inherits the definition of Feature a from Polygon.
3-sided-figure
Feature a (redefined)
4-sided-figure
Inherits the definition for Feature a from 4-sided-figure and the definition of Feature b from Parallel-sides-mixin.
Triangle
Rectangle

Figure 3-3
Class Inheritance

An important issue is resolving feature name conflicts in the inheritance lattice. When a feature is activated in a linear inheritance graph (single inheritance), there may be several feature-class pairs with the same feature name. By convention, the feature class pair chosen for execution is that from the most specific class; so we speak of a *most-specific-first* ordering on feature-class pairs. This is the default assumed in IDEF4.

In the case of multiple inheritance, selection of a feature-class pair for execution from a set of conflicting feature-class pairs is more troublesome. In IDEF4, by default, the conflict is resolved by choosing the most specific feature-class pair in a most-specific-first, topological ordering of the conflicting feature-class pairs in the inheritance lattice. This conflict resolution strategy is employed by most OOPLs, such as CLOS, Smalltalk, Flavors, and C++.

A designer may not always want to use the default strategy employed by IDEF4 (i.e., the dispatching is to be performed according to a different prioritizing scheme). The other prioritizing schemes to consider include 1) a least-specific-first ordering; 2) a depth-first, most-specific ordering as in KEE (KEE Software Development, 1985); 3) a breadth-first, most-specific ordering; or 4) the execution of all the applicable feature-routine pairs, etc. In the last case, the designer may want to specify that the return results are to be combined in a particular manner; this is called a method-combination specification in OOPLs and a method-set-combination contract in IDEF4. If the designer wishes to change the default method-set-combination contract employed by IDEF4 for a certain feature, this must be stated in the method-set contract of the method taxonomy based on that feature. It should be noted that redefinition of the default method-set-combination contract will rarely occur in practice and should be avoided if possible since many OOPLs do not allow it.

From the module point of view, inheritance is a macro-like "virtual copy" operation: all operations and similar features associated with a superclass are automatically inherited by its subclasses, with the exception of those features or operations that are redefined in the subclass. For example, in Figure 3-3, the *Polygon* class defines a feature named *Feature a*. This feature will be inherited in all of its subclasses: *3-sided Figure*, *Triangle*, *4-*

sided Figure, and *Rectangle*. The routine for *Feature a* in both *3-sided Figure* and *Triangle* is identical to that in *Polygon*. Because there is an additional contract on *Feature a* in *4-sided Figure*, it is said to be "redefined" for that class and its subclasses. Since *Rectangle* is a subclass of *4-sided Figure*, the definition applied in *Rectangle* will be the redefinition.

A subclass can see and use any feature of its superclasses, but not vice versa. The notion of inheritance conflicts with the traditional notion of information hiding. This violation, because it is allowed in a controlled and limited way (and in one direction only), is one of the keys to the power of the object-oriented paradigm. A properly structured OOD uses inheritance facilities to minimize duplication of modules. The IDEF4 focus on structuring methods into logical modules or classes helps ensure that the resulting OOD achieves this goal.

In IDEF4, provision is made for the display of the inheritance relationships between classes in the inheritance diagram as shown in Figure 3-4. An inheritance diagram provides information that describes the classes, their features, and any redefinition of features. For example, the reader familiar with IDEF4 syntax can determine that a filled rectangle inherits all features of the classes *Polygon*, *Rectangle*, and *Fill-mixin*. Furthermore, it can be seen that the feature *Perimeter* is an attribute that has been redefined in *Rectangle*. The details of inheritance diagrams will be discussed in Section 4.1.

3.2 Features

A feature is the named representation of a particular characteristic of a class. Features are used to capture the state and behavior of instances of a particular class. A feature may be value-returning and/or side-effecting. For example, the class *triangle* may have a feature called *area* that returns (value-returning) the area of an instance of the class, and a feature called *move* that changes the position on the screen space (side - effecting) of

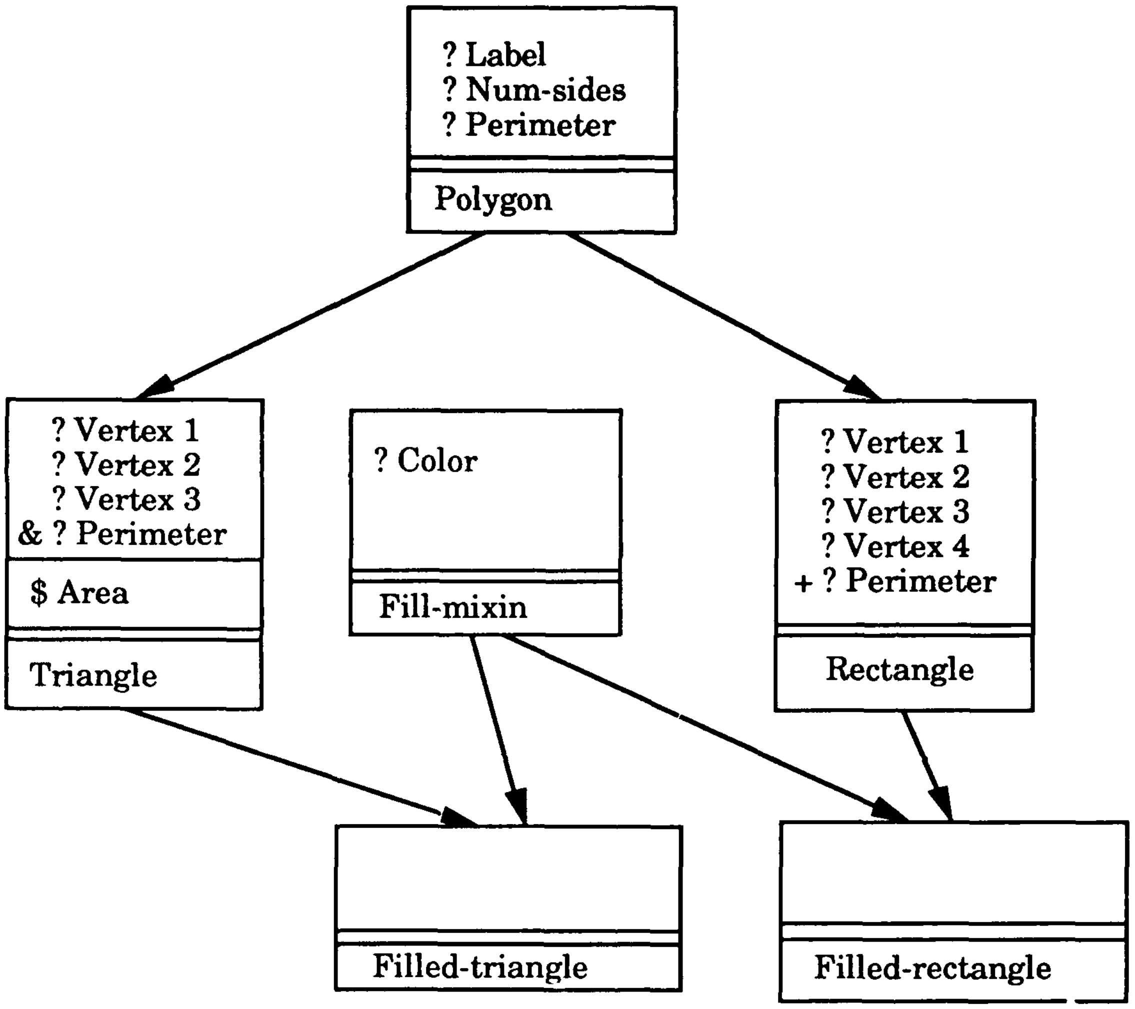


Figure 3-4
Partial Inheritance Diagram for a Graphics System

instances of the class *triangle*. Whether a given value-returning feature is implemented by storage or by computation is functionally irrelevant. It is the type of behavior (or state) that the feature named *area* provides that is of concern. Whether *area* is implemented as a slot or whether the value is computed from other features of the triangle is not necessarily of concern in the initial design stages.

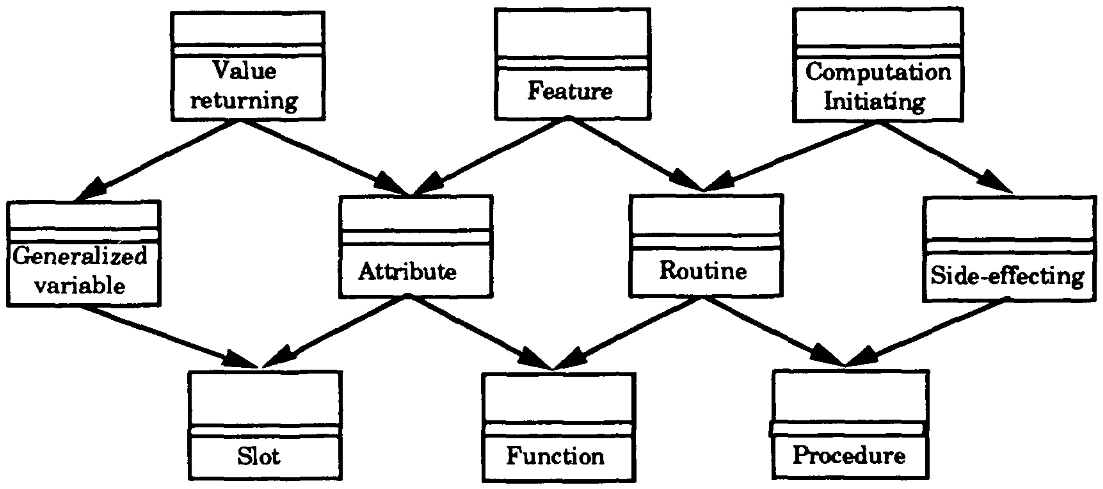


Figure 3-5
Class Feature Inheritance Lattice

This delayed decision-making capability is supported in IDEF4 by 1) subdividing the feature category into attributes (value-returning) and routines (computation-initiating), 2) subdividing the attribute category into slots (value-returning from storage) and functions (value-returning by computation), and 3) subdividing the routine category into functions and procedures (side-effecting). The class-feature inheritance lattice is shown in Figure 3-5 using the IDEF4 inheritance diagram class-box syntax.

3.2.1 The Feature Taxonomy

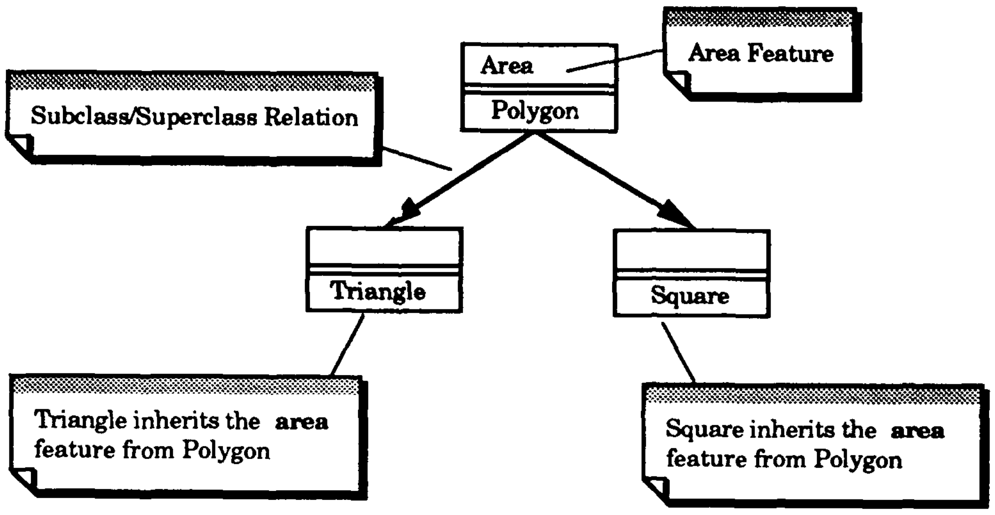
The feature taxonomy allows features to be characterized in general terms initially; then, gradually, to be defined more specifically as the design evolves. For example, a designer might first specify a characteristic of a class as a feature. Then, as the design evolves, the designer can specialize the definition of that feature to an attribute, a routine, a slot, a function, or a procedure.

Figure 3-5 displays an inheritance lattice of the major types of class features that are inherited and their properties. The arrows represent the specialization relation and point from least-specific to most-specific class. At the least-specific (top) level, all class features are grouped together simply as features. The principal defining characteristic of an attribute is that it returns a value when queried. A routine is something which, when appropriately triggered, will initiate computational activity. These are not mutually exclusive categories. The three most specific sets of features (i.e., slots, functions, and procedures) are mutually exclusive. Features classified as functions have characteristics of both attributes and routines; they return a value by computing it whenever queried. Features classified as procedures have characteristics of routines that do not return values. They are computational activities that have only side effects, though strictly speaking a no-op (no operation) procedure might be useful in some contexts. Slots are attributes that are not routines. They are similar to generalized variables in Common LISP (List Processing). That is, they can return values when queried because they have access to some sort of storage facility, ultimately computer memory, which can hold values for retrieval. For a generalized variable, the operation of assignment, or changing the value held for retrieval, makes sense. This is the intention implied by the IDEF4 slot-type of feature as well. The three most specific feature-type categories (slots, functions, and procedures) are mutually exclusive and form a partition of the set of all class features in an OOD.

3.2.2 Feature Inheritance

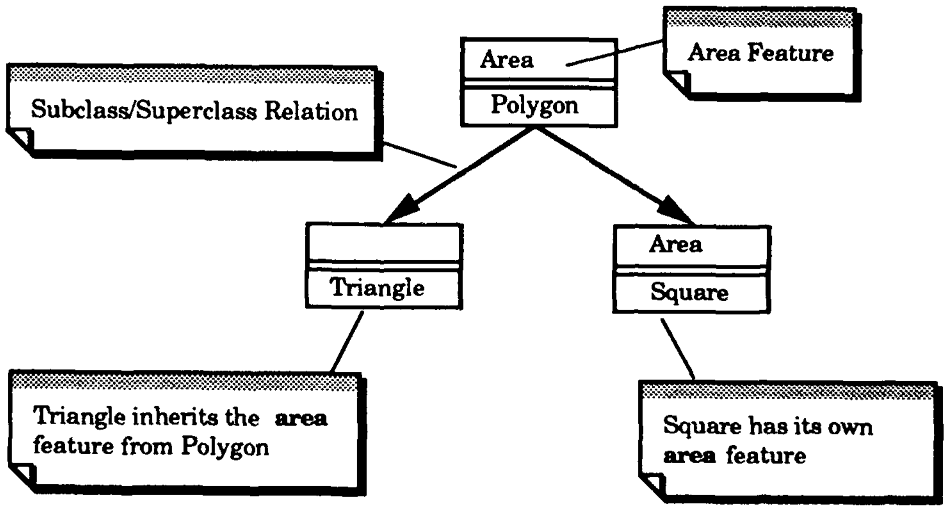
All features associated with the superclass are automatically associated with the subclass through class-inheritance mechanisms. A subclass can see and use any feature of its superclasses, but not vice versa. Figure 3-6, for example, shows that the class *Polygon* has a feature called *Area*. Subclasses of *Polygon* such as *Square* and *Triangle* inherit the area feature from *Polygon*.

If the *Area* feature is implemented as a calculation, then using the general area calculation of the polygon for a square would be inefficient. Therefore, it might be desirable to define an *Area* feature for *Square* that uses the more efficient, specialized calculation for instances of squares. This specialized calculation would be invoked instead of the more general method. Thus, the more specialized method "shadows" or redefines the more specific method. Figure 3-7 illustrates a case in which the *Area* feature of the class *Square* shadows or redefines that of the class *Polygon*. The *Triangle* class continues to inherit *Area* from *Polygon*.



Area Feature
Area
Polygon
Subclass/Superclass Relation
Triangle
Square
Triangle inherits the **area** feature from Polygon
Square inherits the **area** feature from Polygon

Figure 3-6
Inheritance of Area from Polygon by Triangle and Square



Area Feature
Area
Polygon
Subclass/Superclass Relation
Triangle
Area
Square
Triangle inherits the **area** feature from Polygon
Square has its own **area** feature

Figure 3-7
The Area Feature of Square Shadows that of Polygon

3.2.3 Feature/Class Taxonomy

It is important to classify features with respect to each object class in which they are present. This second classification scheme allows the designer to indicate the way in which he intends to associate a feature with a class (i.e., defined in the class, redefined in the class, or an inherited feature in the class). The taxonomy for this secondary classification scheme is depicted in Figure 3-8.

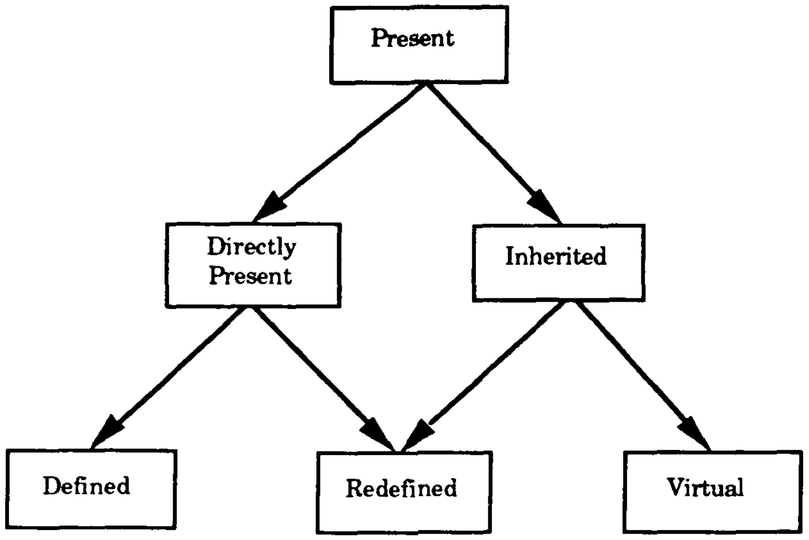


Figure 3-8
Feature/Class Taxonomy

Using this classification scheme, a feature that is associated in any way with a class is present in the class. Those features whose names are displayed in the class box of a class *A* are said to be directly present in *A* (relative to a diagram). Those features present in a superclass *B* of *A* are considered to be present in *A* as well, and are inherited features of *A*. Features of *A* that are both directly present and inherited in *A* are redefined in *A*. They are directly present because the class box is giving additional or revised information about them that was not present in the superclasses of *A*. Features that are directly present but not inherited in A are said to be defined in *A*; those that are inherited but not directly present are said to be virtual in *A*.

The matrix in Figure 3-9 categorizes the *Area* feature of the class hierarchy described in Figure 3-7. Figure 3-9 shows the secondary classification for the *Area* feature with respect to each class in the hierarchy as 1) present, directly present, and defined in the *Polygon* class; 2) present, inherited, and virtual in the *Triangle* class; and 3) present, directly present, inherited, and redefined in the *Square* class.

3.2.4 Feature Types

Value-returning features have a type associated with them which specifies the type of the value returned. The type can be a canonical type such as integer or real, a class, or structured collections of classes.

In the IDEF4 discipline, all attributes have a return type, defined as the type of their return value. From a design management point of view, feature types provide an implicit indication of the relations between classes. This information is not visible in the inheritance lattice. Experience has shown, however, that management of these interconnections is critical to the development of large object-oriented systems. Through the study of the feature types, the programming team can communicate intended relations.

Class	Present	Directly Present	Inherited	Defined	Redefined	Virtual
Polygon	X	X		X		
Triangle	X		X			X
Square	X	X	X		X	

Figure 3-9
Area Feature in Polygon Class Hierarchy

Figure 3-10 illustrates feature-type relations. It shows four classes 1) *Movable*, 2) *2-Wheeled,* 3) *Real*, and 4) *Wheel*. The *Movable* class defines two features (*X* and *Y)*, which return real numbers that indicate the position of instances of movable objects. The class *2-Wheeled* inherits the features of a movable class through the subclass/superclass relation. The *2-Wheeled* class defines features (*Wheel-1* and *Wheel-2)* that return instances of the class *Wheel*. The *Wheel* class has a *Diameter* feature that returns values of type *Real*.

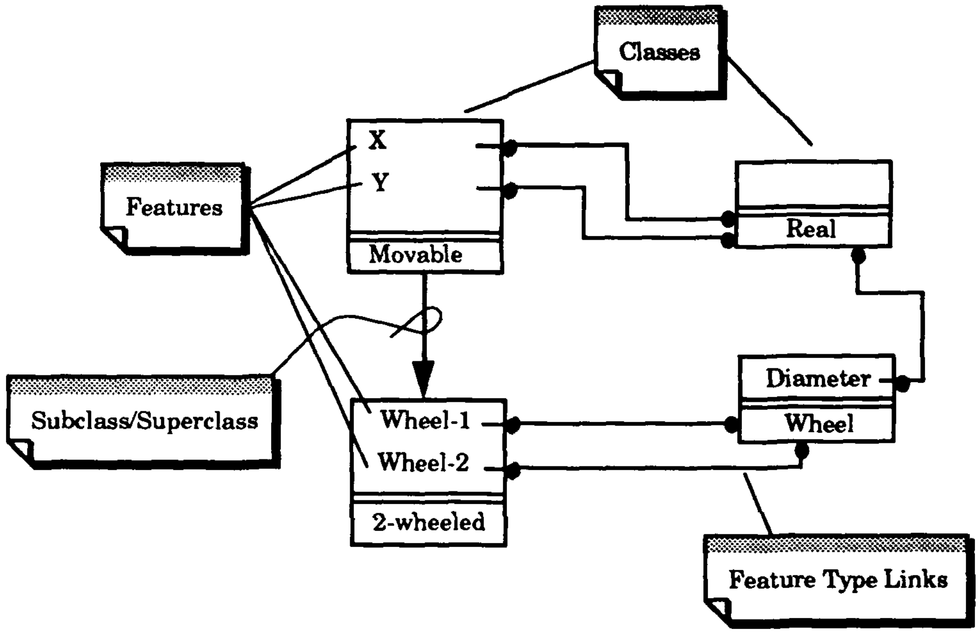


Figure 3-10
Feature Types

3.2.5 Class Features

Figure 3-11 shows type links and inheritance links of a *Shape* class-inheritance lattice. The base class of this inheritance lattice is the *Polygon* class, which is inherited by the *Convex* class, which in turn is inherited by the *Square* and *Triangle* classes. In instances of *Polygon* and *Convex*, the *Nr-sides* feature returns integer values, whereas in *Square* and *Triangle*, the *Nr-sides* feature is restricted to returning a fixed value for all instances of the class. The features that have this restriction are called class features. This restriction may be expressed as a class-invariant constraint (i.e., a constraint that holds for all instances of a class) or as a constraint on the contract that the feature fulfills.

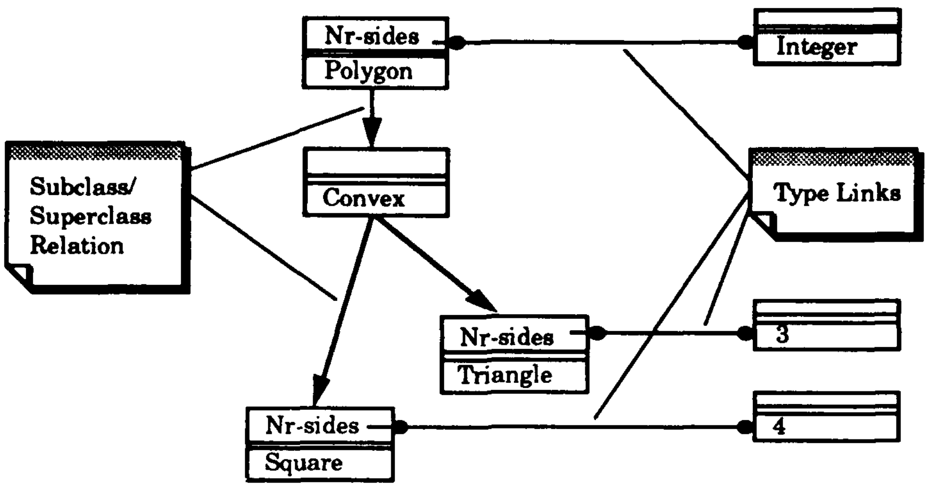


Figure 3-11
Class Features

An example of a class-invariant constraint that satisfies this restriction is:

For-all (x)(triangle x) -> (Nr-sides x) = 3

For-all (x)(square x) -> (Nr-sides x) = 4.

Some object-oriented languages such as CLOS can represent the class feature directly as a class attribute.

3.3 Methods

As shown in the discussion on class features, a class may have features that define the behavior of its instances. The behavior of an object is how it responds to a message or generic function in terms of changing its state, altering its environment, acting on other objects, and returning values. The features that define behavior are computation-initiating; thus, by definition, they are routines. The function and procedure feature classes are subclasses of the feature class routine; more specifically, they inherit the computation-initiating property from the feature-class routine. It is convenient to speak of behavior-defining features as routines. Each routine must be defined in .. ass so that a routine may be uniquely identified by a generic routine-class pair. In this document, the term "routine-class pair" will be used in place of "generic routine-class pair" when it is obvious that we are referring to a specific behavior defined in a particular class. The term "routine" will be used instead of "generic routine" when it is obvious that we are referring to the class of behavior. In case of possible ambiguity, the unabridged form of the terms will be used.

3.3.1 Methods in Design vs. Methods in Programming

Routines may be attached as features to more than one class and may behave somewhat differently depending on the class of objects upon which they perform their computations. In OOPLs these routine-class pairs are implemented by a single method that provides the required behavior. The notion of a method in IDEF4 design is not the same as the usual notion of a method from an object-oriented language point of view. In object-oriented programming, a method is an executable piece of code which algorithmically specifies the computation to be performed by means of a set of instructions. In IDEF4, methods are defined by the contract that they must fulfill. In fact, in IDEF4 we do not refer to individual methods; rather, we refer to a set of methods, any of which can fulfill a specified contract. In other words, we refer to the contract for a programming language method rather than the code for the method.

In general, more than one method will be associated with a routine, even in a programming language. The same routine may have different methods in different classes. In an OOPL however, only one method will be associated with each pair of a routine and a class. In Figure 3-12, the routine *Area* is a member of two routine-class pairs, *Area: Polygon* and *Area: Triangle*. In a programming language, a different method would be defined for each of these, such as *Get-area-of-polygon* and *Get-area-of-triangle*. In IDEF4, on the other hand, individual methods are not represented. Thus, in IDEF4, *Get-area-of- polygon* and *Get-area-of-triangle* would refer to method sets and their related contracts illustrated in the *Area* method taxonomy. The routine-class pair specifies a method set defined by an associated contract. Any member of this set satisfies, and would satisfactorily implement, the routine for the class. The idea in IDEF4 is to describe or design the behavior, not program the behavior.

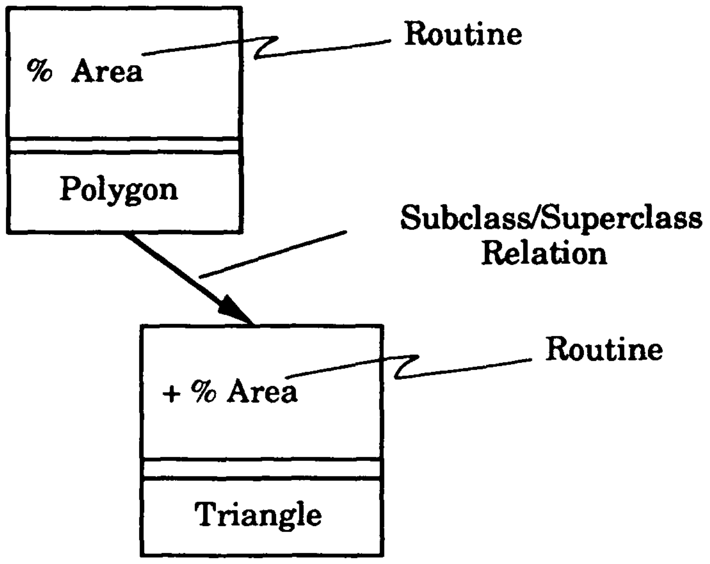


Figure 3-12
Same Routine in Different Classes

3.3.2 Method Sets

The method set can be thought of as a characteristic function defined by a set of constraints that pick out a set of possible correct implementations. The set of constraints -- the defining characteristics for methods in the method set, is called a method set contract. IDEF4 treats method sets at the conceptual level; therefore, in a design, the concern is with the definition of the contract, rather than individual methods within the set. The expression *method set* is often used to refer to the constraints characterizing a set of methods. These characterizing constraints are required for any implementation. In IDEF4, the constraints are satisfied by any method written to implement the behavior of a routine-class pair. This defined set of constraints is the method set contract discussed in Section 4.2.1.

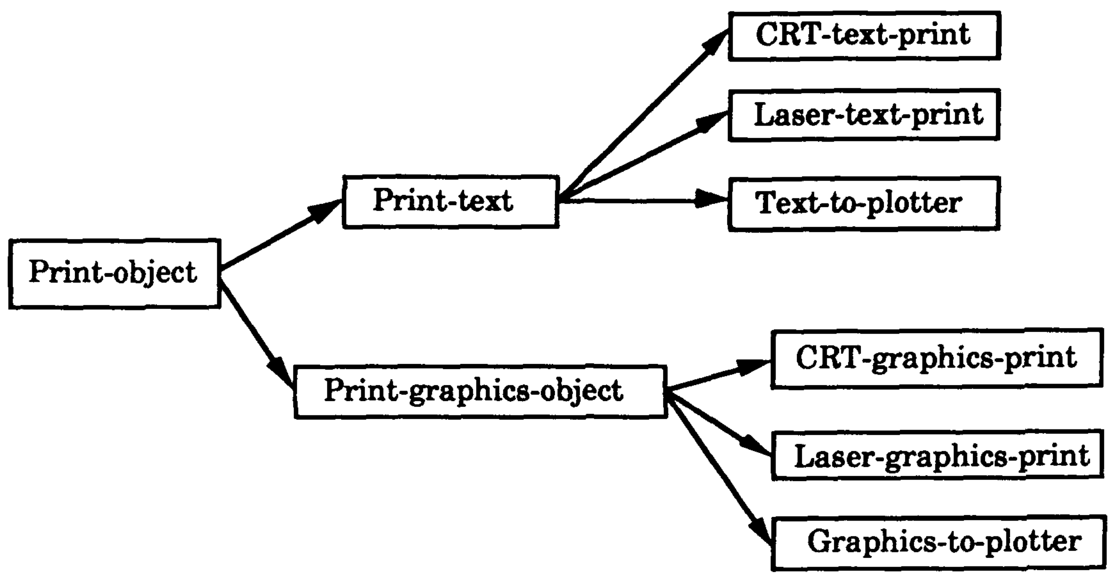


Figure 3-13
Method Set Relations

For example, the class-invariant constraint on the feature *Identity-number* of type integer in the class *Soldier* may be expressed as:

"The Identity-number feature in the Soldier class must be a unique integer over all instances of the class soldier."

More formally, this might be written as the following constraint that specifies that no two soldiers may have the same identity number:

```
For-all(x y)  (soldier x)^(soldier y)^(not-equal x y)
        ^(not-equal(identity-number x)(identity-number y)).
```

Another example of a class-invariant constraint would be one that applies to a doubly linked list.

A constraint for the contract of the method *Pop* in the class *List* might be specified as:

> "The Pop operation is inappropriate when the list it is being applied to is empty. When this operation is attempted on an empty list, the operation should return a nil to indicate an exception condition."

More formally, if the Pop operation is applied to a list and the list is empty, then the result is nil:

```
For-all (x y)     (invocation pop x)^(list y)^(empty y)
        ->
            (return x y nil).
```

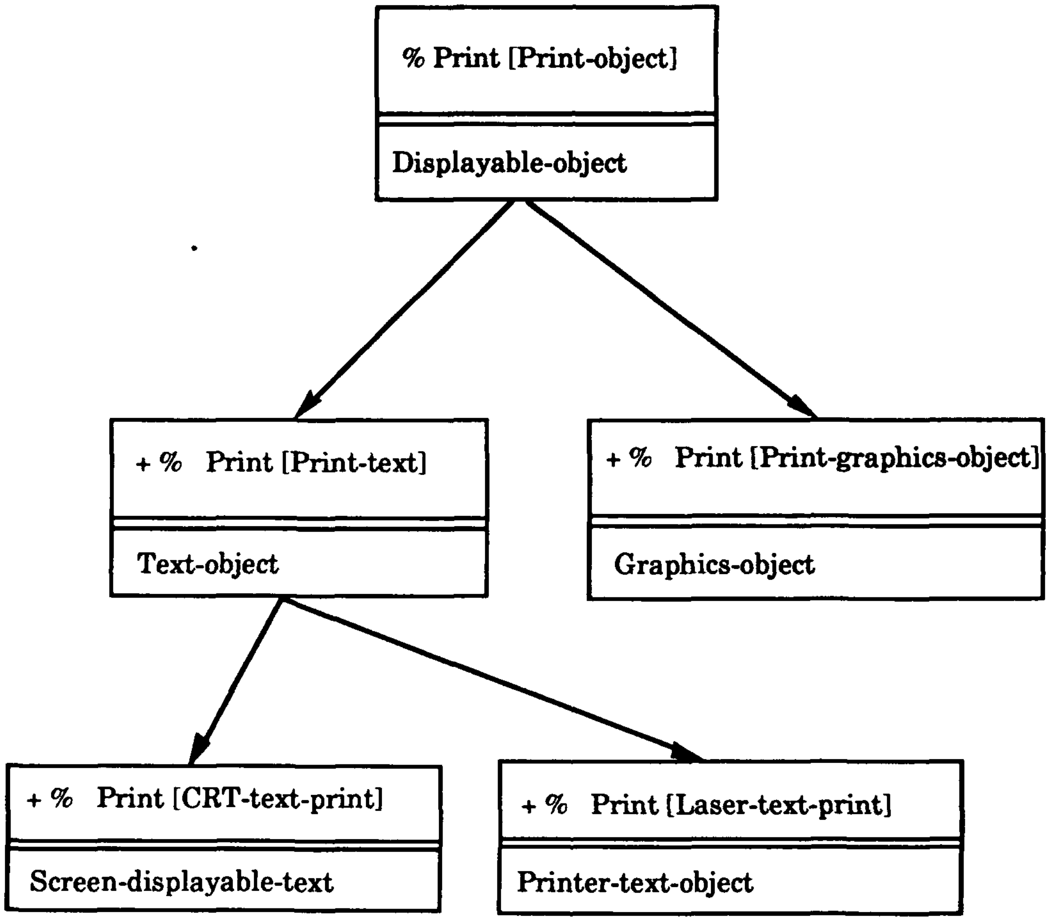


Figure 3-14
Class Structure with Method Redefinition

4.0 Diagrams

Five types of diagrams and two types of data sheets are used to create an IDEF4 design model. The diagram types are grouped into the following two submodels.

1. Class Submodel
    - Type Diagrams
    - Protocol Diagrams
    - Inheritance Diagrams
2. Method Submodel
    - Method Taxonomy Diagrams
    - Client Diagrams

These submodels are connected via dispatch mapping, which may be displayed in both inheritance diagrams and method taxonomy diagrams. The discussion of each diagram type will contain a concise description of all the symbols that may be used in the diagram.

4.1 Class-inheritance Diagrams

Class-inheritance diagrams are part of the IDEF4 class submodel. They are used to graphically display the class hierarchies of a system.

4.1.1 Class-inheritance Diagram Symbol Set

In this section, we discuss the symbols that are used to create a class-inheritance diagram. These consist of the class box, symbols for describing the features of the classes, and arrows that display the superclass to subclass relationship.

In the class box, the name of the class itself appears at the bottom of the box below a double horizontal line with the first letter capitalized as shown in

Figure 4-1. The names of the features that are directly present appear above the double horizontal line. Features may also be divided into two groups above the double horizontal line by the addition of a single horizontal line (the export line). The export line separates the directly present, public features of a class from the directly present, private features of the class. Public features are those that appear above the line and are said to be exported from the class. Private features are not exported, that is, they are only visible to the class and its subclasses. Virtual features are public or private according to their status as public or private in their source class. In fact, virtual features have exactly the same characteristics that they have in the source class. The source classes, if there is more than one, must agree on the characteristics of a virtual feature.

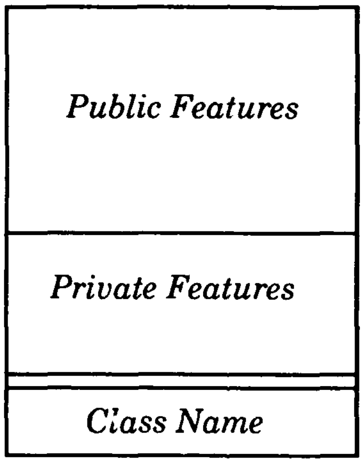


Figure 4-1
Class Box

If a feature is public in a class, then its presence in that class is visible to and usable by all classes in the IDEF4 model. If a feature is private in a class *A*, then its presence in *A* is visible to all of *A*'s subclasses, but not to any other classes (unless it is otherwise made visible to them). If no export line appears, then all directly present features of the class are public. If an export line appears, but there are no features above it, then all directly present features of the class are private. (This might happen if the class were intended solely as a "mixin" to be combined with other classes via multiple inheritance.) It is also permissible for a class not to have any directly present features; in this case, the export line must not appear. In the IDEF4 class box for the *Triangle* class featured in Figure 4-2, *Name*, *Vertex 1*, *Vertex 2*, *Vertex 3*, and *Area* are public features; *Area-internal* and *Area-flag* are private features.

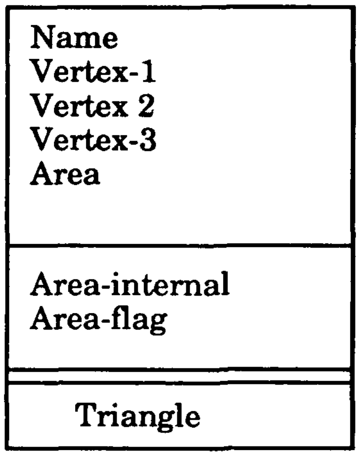


Figure 4-2
Triangle Class Box with Features

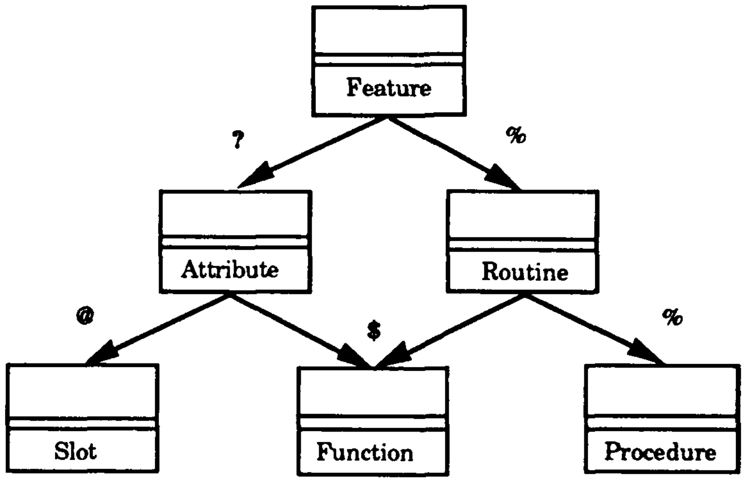


Figure 4-3
Feature Taxonomy Showing Specialization Symbols

IDEF4 features may be routines, attributes, functions, procedures, or slots. Initially, the designers will not make a decision on how the features are implemented, only that particular features will exist (see Figure 4-2). This design practice is perfectly acceptable and is likely to be the case in the early definition of class features. As the development of the design continues, the designer may choose to specialize the feature classification. (The design may evolve in such a way as to result in certain features being transformed from a more specific class to a more general class.) When the designer wishes to commit to the type of intended feature, a feature symbol may be added to the left of the feature name (see Figure 4-4). The feature specialization symbols are shown on the feature class-inheritance diagram shown in Figure 4-3.

The feature symbols are:

	Feature - No symbol to the left of the feature name indicates that feature is present but its implementation has not been determined.
%	Routine - The % symbol indicates the feature will, when appropriately triggered, initiate some computational activity. It is not known or specified whether a value will be returned. These details may be left to the implementors of the design.
?	Attribute - The ? symbol indicates that the named feature will return a value when queried. This value may be a stored value or the return value from some computation.
$	Function - The $ symbol indicates that the named function will return a value by computing it whenever queried. A feature that is implemented as a function has the characteristics of both an attribute and a routine.
#	Procedure - The # symbol indicates that the named feature is a routine that does not return a value. It is included for the side effect(s).
@	Slot - The @ symbol indicates that the named feature is a generalized variable. Slots can return values when queried because they have reading and writing access to the storage location in which the slot value is stored. Operations for value retrieval and assignment of the value

held for retrieval will be associated with this type of feature.

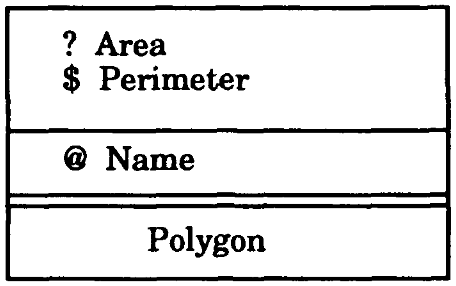


Figure 4-4
Polygon Class with Feature Symbols

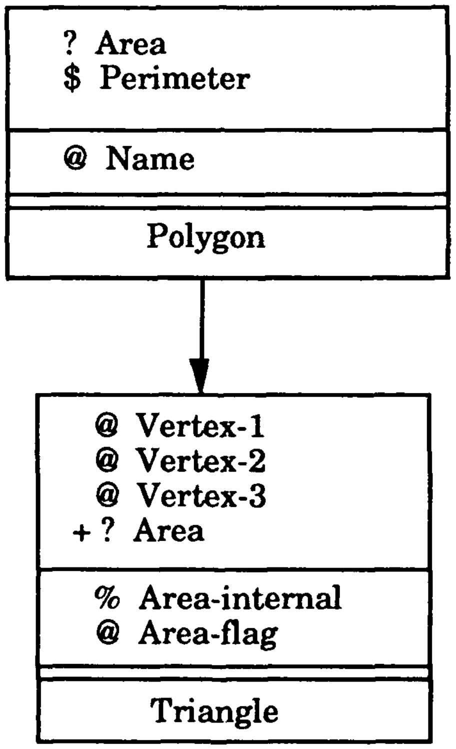


Figure 4-5
Triangle Class a Subclass of Polygon

The feature symbols indicate how the designer expects a particular feature to be implemented. For example, in Figure 4-4 the *Polygon* class has three features: *Name*, *Area*, and *Perimeter*. The *Name* feature is to be implemented as a slot, the *Area* feature as an attribute, and the *Perimeter* feature as a function.

In Figure 4-5, the superclass/subclass relationship between the *Polygon* and *Triangle* classes is illustrated. This relationship is defined by an arrow that points from the superclass *(Polygon)* to the subclass *(Triangle).* Subclasses of the *Polygon* class will inherit all *Polygon* features and implementations. This characteristic of inheritance is not always what the designer intends. For example, the *Triangle* class (Figure 4-5) is a subclass of the *Polygon* class, but the algorithm for calculating the area for a general polygon is too general for a triangle. For this reason, the designer must indicate that there is to be a redefinition of the *Area* feature for an object of type *Triangle.*

IDEF4 provides symbols to be used in a subclass if an inherited feature is redefined. One or more of the following auxiliary feature symbols will appear to the left of the redefined feature name (and feature symbol, if any) showing how the feature has been redefined.

&	Additional Contract - This symbol indicates that an additional constraint added to the method contract associated with the named feature (e.g., a pre-condition or post-condition).
+	New Method - This symbol indicates that the named feature represents another routine with the same name as the one inherited from the superclass (as in the Triangle type in Figure 4-5). The additional constraints placed on the routine require that a new contract be stated in its entirety.
!	New Taxonomic Specification - This symbol indicates that a feature is redefined more specifically as a function or slot in a subclass, and perhaps differently in different subclasses.
^	Now Public - This symbol indicates that a previously private feature has been made public and its presence in the class (and therefore in all succeeding subclasses) is visible to and usable by all classes in the IDEF4 model.

The name of the feature that is redefined as public should be placed above the export line.

* Now Private - This symbol indicates that a previously public feature has been made private and its presence in the class is visible to all subclasses of that class, but not to any other classes in the IDEF4 model. The name of the feature redefined as private should be placed below the export line.

These symbols for class boxes, feature identification/redefinition, and arrows are combined to form the class-inheritance diagrams.

4.1.2 Understanding a Class-inheritance Diagram

Now that we have discussed the basic symbol set for inheritance diagrams we will discuss the graphical diagram in which this information is displayed. The basic IDEF4 language component for this purpose is the inheritance diagram. The inheritance graph of a particular IDEF4 model is simply a single, maximal inheritance diagram that shows all the classes and direct inheritance relationships. An inheritance diagram of this size would have little practical use if actually drawn, but it is useful to keep in mind as a way of imagining the full scope of inheritance diagrams. They differ from the full inheritance graph only by omission. The full inheritance graph could also be stored in computer memory to allow automated creation of diagrams for video display.

In IDEF4 inheritance diagrams, classes are displayed using the class box notation; inheritance is represented by an arrow pointing from one class box (superclass) to another (subclass). In Figure 4-6, the arrow from *Polygon* to *Triangle* indicates that *Triangle* is a subclass of *Polygon*. Inheritance is also transitive: if *Filled-triangle* is a subclass of *Triangle* and *Triangle* is a subclass of *Polygon*, then *Filled-triangle* is a subclass of *Polygon*. With respect to a given IDEF4 diagram, inheritance that is shown directly by a single arrow from class box *Polygon* to class box *Triangle* is called direct inheritance. (The phrase "with respect to a given IDEF4 diagram" is used because not all IDEF4 diagrams are obliged to show all

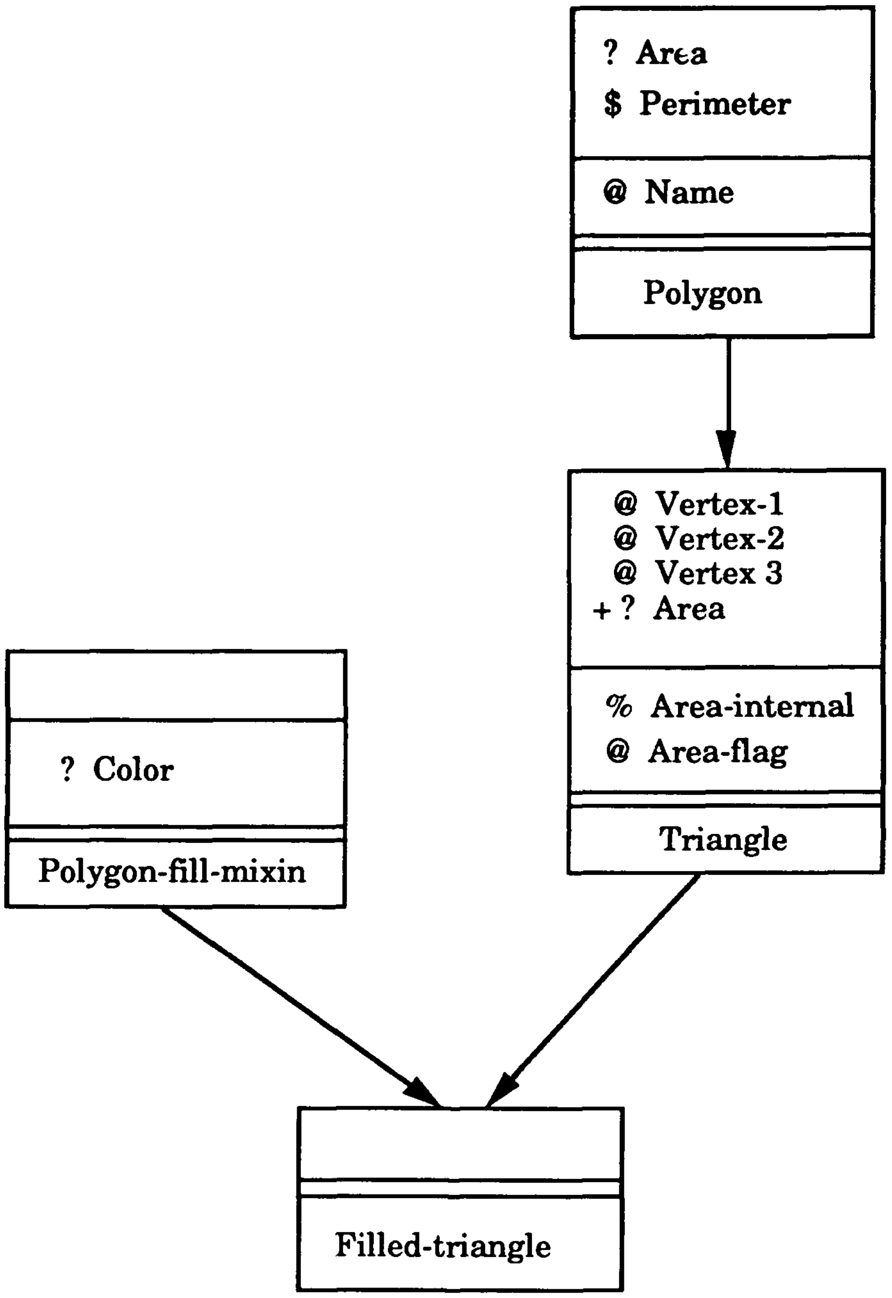


Figure 4-6
Partial Inheritance Diagram for a Polygon Class

connecting links between two classes.) *Polygon* is said to be a direct superclass of *Triangle*, and *Triangle* a direct subclass of *Polygon*.

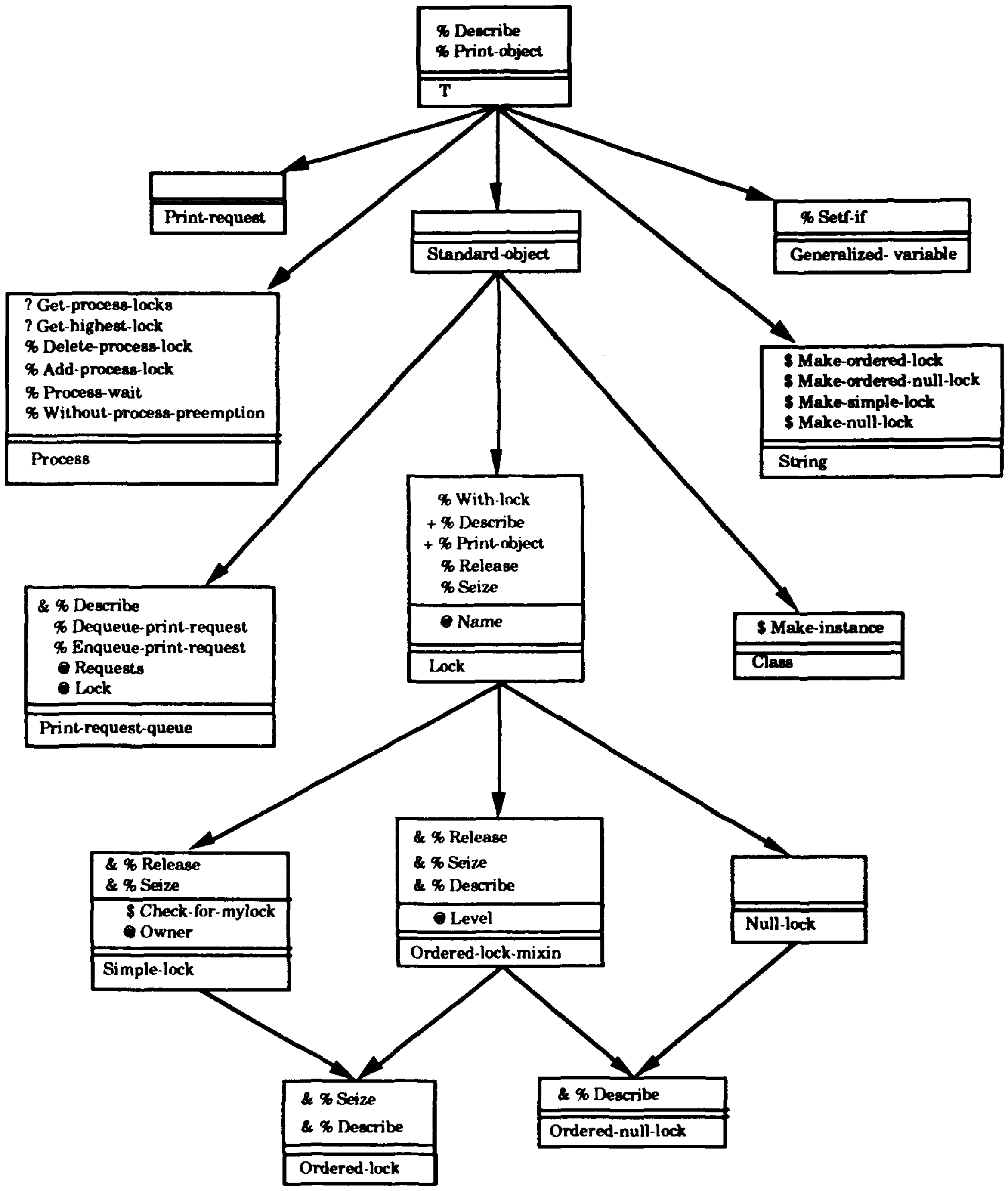


Figure 4-7
Inheritance Diagram - Network Manager System

(Based on the extended example of CLOS programming, Chapter 3 (Keene, 1989).)

If inheritance is not shown by a single arrow and can only be inferred from the presence of a chain of arrows, using transitivity—as in Figure 4-7 between *Standard-object* and *Simple-lock*—the terms indirect inheritance, indirect superclass, and indirect subclass are used. ((Keene, 1989) provides a detailed description of the system modeled in this diagram.)

This graphical approach for describing the class hierarchy structure was designed to maximize the amount of key information displayed in a minimum amount of space. The class-inheritance diagram identifies the features of the classes and subclasses that are displayed in the boxes, and also reveals details about the implementation of the features and their visibility within the system. In Figure 4-7, the more specific class, *Standard-object*, has only inherited features from the class *T*. Since these inherited features are not redefined, they do not appear in the *Standard-object* class box. *Lock,* which is a direct subclass of *Standard-object* and an indirect subclass of *T,* also inherits the features *Describe* and *Print-object*. In *Lock,* the plus sign (+) preceding *Describe* and *Print-object* indicates that these features have been redefined with constraints that will shadow some of the constraints in the superclass definition.

If the intent had been to illustrate that the redefinition consisted of the addition of pre- or post-conditions to the set of constraints for the superclass routine, an ampersand (&) would have been used. The directly present features *With-lock*, *Release*, and *Seize* are all *defined* as routines in the *Lock* class and are public. The @ symbol prefixed to the directly present feature *Name* is defined as a slot and its appearance below the export line indicates that it is private.

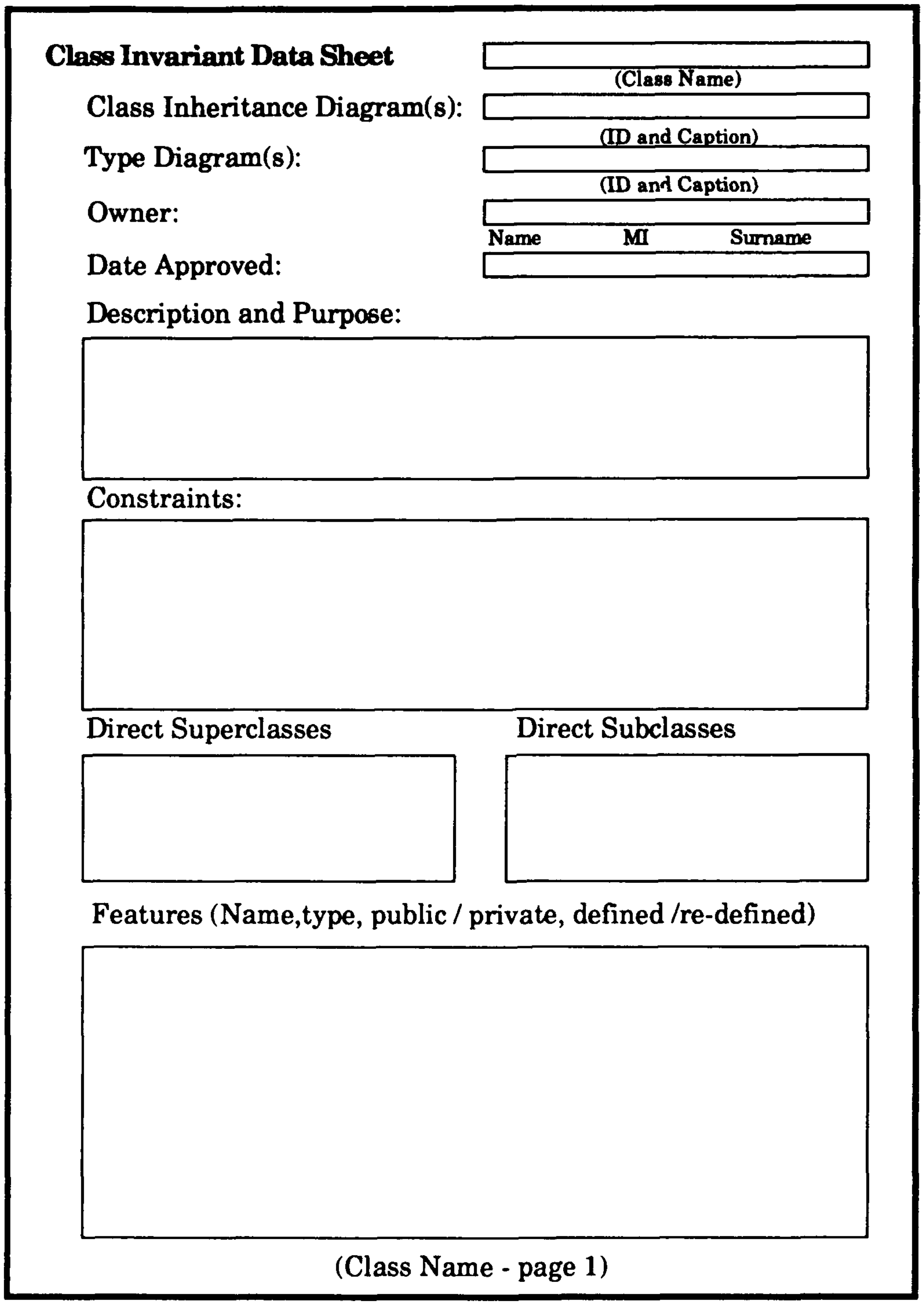
4.1.3 Class-invariant Data Sheets

In IDEF4, each class has an associated specification of the definition of that class. This definition is entered into a CIDS because it captures (in specification language form) the designer's intentions for the property values or object relations that individual instances of that class must possess. The CIDS allows refinement of the definition of a class beyond the listing of the class features and relations that are shown on the various IDEF4 diagram languages. The CIDS can be considered to be attached to individual class boxes in an inheritance diagram (see Figure 4-8). The CIDS gives further information about the objects in a class, perhaps relating various distinct features in textual form. This information taken as a whole makes up a *class-invariant* which must be maintained as true of all objects in the class always. A CIDS for *Linked-list* might include the constraint that the list constructed out of the first element and the rest of the list is identical to the original list.

equal(cons(first(x),rest(x)),x)

The class-invariant constraints are inherited just as features are. The combining principle is that of logical conjunction. A class invariant is the logical conjunction of what appears on its own CIDS and what appears on those of all its superclasses. (The term class-invariant can also be loosely used for the contents of a single CIDS, or even just for a single constraint appearing on such a sheet, like the one for *Linked-list* above.)

A CIDS is more than just a vehicle for listing the constraints on a class. One purpose of CIDS is to provide documentation for those who will maintain the installed system as well as those who will implement the design. IDEF4 design documentation is centered on the classes defined for the system; therefore, CIDSs include the numerical identifiers for referencing the inheritance diagrams and type diagrams in which the class appears. The names of directly present features of a class are provided to allow access to the routine-class pairs around which the other design components are organized. The definition of each feature is included with the feature name. Virtual features are accessed via the listing of the direct superclasses of the named class. The CIDS for a class can be more than one 8-1/2 in. by 11 in. page because they contain so much information. Provision is made for this by including page numbers. CIDS are not expected to be complete even when the class description has been



Class Invariant Data Sheet (Class Name)

Class Inheritance Diagram(s): (ID and Caption)

Type Diagram(s): (ID and Caption)

Owner: Name MI Surname

Date Approved:

Description and Purpose:

Constraints:

Direct Superclasses

Direct Subclasses

Features (Name,type, public / private, defined /re-defined)

(Class Name - page 1)

Figure 4-8
Class-invariant Data Sheet Form

completed. Some information in a CIDS may not be available until the system design is near completion. For example, not all the inheritance diagrams nor type diagrams in which the class appears may be known at the time the constraints and features are determined. Thus, just as other components of the design evolve over the design process, these data sheets will evolve as well.

Figures 4-9 and 4-10 illustrate the use of CIDS. The CIDS illustrated in Figure 4-10 can be considered to be attached to the *Lock* class (Figure 4-9) which appeared in the class-inheritance diagram shown in Figure 4-7. The information at the top of the sheet identifies the name of the class and other relevant bookkeeping types of data. The inheritance diagrams in which the class is drawn will be listed by ID and Caption. This provides a link to CIDS and allows the implementor access to a visual representation of the class's place in the system class hierarchy. The description components allow identification of the types of objects described by the class and any other information related to its reason for being.

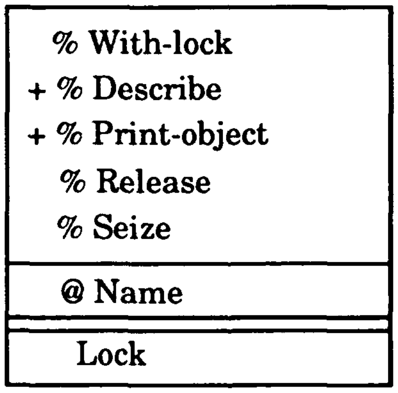


Figure 4-9
Network Manager System *Lock* Class

The primary component of CIDS is the set of constraints on the implementation of the class. These can be stated in plain English (see Figure 4-10), first-order logic, or some other language suitable for expressing constraints. In the class hierarchy, every class inherits the class-invariant constraints of its superclass. By providing the list of the

Class Invariant Data Sheet	Lock (Class Name)
Class Inheritance Diagram(s):	I1 - Network Manager System (ID and Caption)
Type Diagram(s):	T1 - Network Manager Locks (ID and Caption)
Owner:	Jack Q. Designer Name MI Surname
Date Approved:	3/15/91

Description and Purpose:

Lock is the basic lock type. It is used as the superclass for all lock types.

Constraints:

A lock can only be owned by one process.

Each shared resource has only one lock.

A lock can not be released by any process other than its owner.

Direct Superclasses	Direct Subclasses
Standard-Object	Simple-Lock Ordered-Lock-mixin Null-Lock

Features (Name,type, public / private, defined /re-defined)

With-lock: (routine, public, defined) This routine is a syntactic extension to the locking protocol.

Describe: (routine, public, re-defined) This routine specializes the object print routine for locks and overrides the system-defined describe routine.

Print-object: (routine, public, re-defined) This routine specializes the object print routine for locks and overrides the system-defined print-object routine.

Release : (routine, public, defined) When called, this routine will release a lock that is owned by the process that is attempting the release.

Seize : (routine, public, defined) The generic Seize routine for all locks.

Name: (slot, private, defined) This feature is the name attribute for a lock.

(Lock - page 1)

Figure 4-10
Class-invariant Data Sheet for *Lock*

class's direct superclasses, the implementor can locate inherited constraints. The listing of the direct subclasses provides a link to those classes which will inherit the constraints of the class. The listing of the superclasses and subclasses provides more than constraint traceability. If class *Lock* were to be deleted or modified, these two lists would provide those modifying the system design with a means of quickly tracing which classes in the system would be affected by the change.

The final part of the CIDS is the list of directly present features of the class. Inherited features can be found by examining the CIDS for the superclass of the class Lock. For each directly present feature of Lock, the CIDS will contain the name, type, etc., that can be found in the class-inheritance diagram. In addition, the method set whose contract the implementation of the feature is to satisfy and the definition of the feature are also contained in CIDSs. This feature definition in a CIDS is the only place in the design that a textual definition of a feature is provided.

4.2 Method Taxonomy Diagrams

In this section, we will discuss the method taxonomy diagrams and CDSs that are associated with the method sets in the diagrams. The method sets into which methods are grouped by contracts form a taxonomy (grouping) which is at least partially independent of the way in which routines are grouped by the class-inheritance graph. IDEF4 introduces, as part of an IDEF4 model method submodel, method taxonomy diagrams to represent the groups of related method-set contracts within the design. Because an understanding of method-set contracts is central to the understanding of method taxonomy diagrams, they will be addressed first.

4.2.1 Contract Data Sheets

Strictly speaking, a method in an OOPL is described by its code; in IDEF4, a method is any implementation that satisfies the contract in the method set. In IDEF4 terminology, a method set is completely determined by its contract, and logically equivalent contracts pick out identical method sets.

Although this may sound like a method contract is a specification, it is in fact much more. A method contract is an abstract description which defines a class of routines or a method set. It is a declarative statement of the intended effect of the methods in the method set. For a non-side-effecting function, the contract would state the relationship between the function argument list and the corresponding return values. For a procedure or a side-effecting function, the contract would also define how the method set changed the entire state of the world when given an argument list and a prior state of the world. The method contract may in some situations contain algorithmic restrictions such as, "The move method will execute an erase method followed by a draw method." A contract might also give other information about the intended nature of the method-set (e.g., its time complexity). In addition, one would expect to find statements of actions that should occur before a method is invoked and what actions should occur after the method has performed its task.

A method-set contract provides the constraints that are specified to hold for all the implemented methods that would be members of the set. Just as the class-invariant holds for all members of a class, the contract for a method set is an invariant (i.e., it holds for all implementations which are members of the set). For example, consider the class hierarchy shown in Figure 4-11. The *Area* routine in the *Polygon* class specifies that the methods for calculating the area would calculate the area for any polygon. However, when the polygon is a triangle or a rectangle, there are more efficient methods for calculating the area. The class hierarchy shows that the *Area* feature is redefined in each of these subclasses. The method-set contract for these redefinitions would specify the additional constraints to which the designers expect the method sets to adhere. In Figure 4-11, the auxiliary symbol "+" indicates that the additional constraints are such that an entirely new contract would be written for the *Area* routine for both the *Triangle* and *Rectangle* classes. Each time a routine is defined or redefined for a class, the designers must specify the constraints that the methods which implement that routine are expected to satisfy. This set of constraints forms a method-set contract.

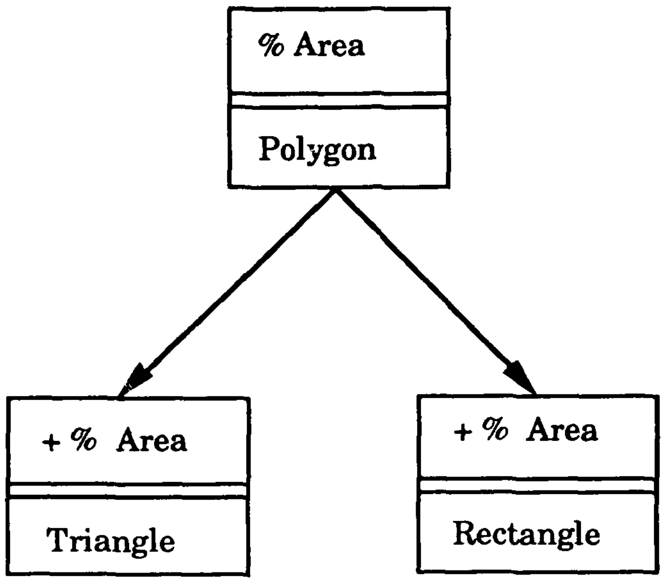
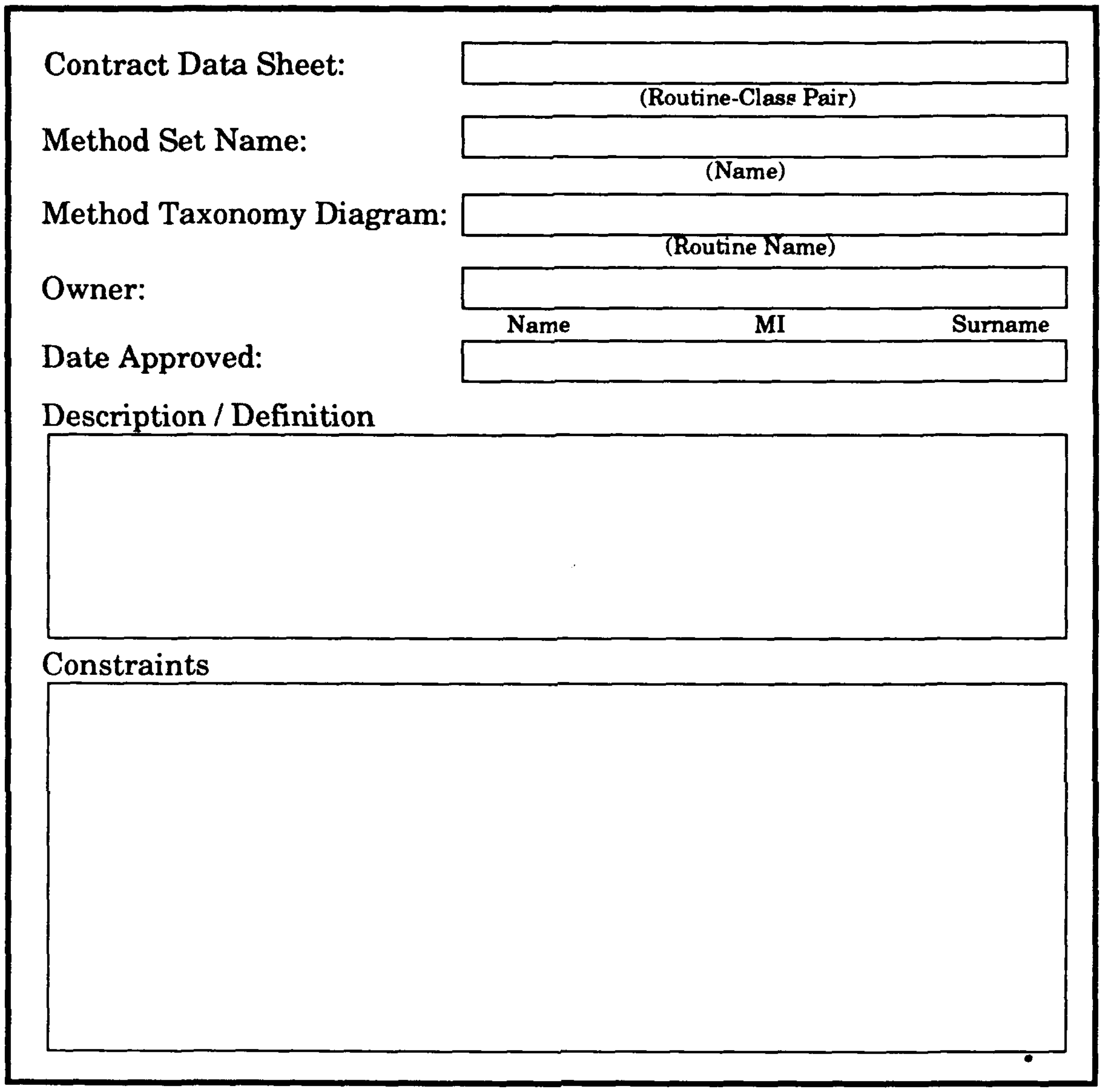


Figure 4-11
Partial Inheritance Diagram with Area Feature

Poor communication and coordination is a primary cause of loss of productivity in software development. All too often, the reason the software demonstration fails to work as expected is that another programmer has altered it without record or explanation. The CDS associated with a method set is one means of ensuring that the code produced by all the programmers conforms to the same expected system behavior.

In IDEF4, contracts are recorded on CDSs associated with the method set as shown in Figure 4-12. One aim of using contracts is to facilitate communication and coordination between designers and implementors in a large software project. The CDSs in the final documentation are organized alphabetically by the routine name followed by class name. Given a CDS, it is easy to reference all other IDEF4 diagrams by using the routine name and/or class name. The routine name is used for referencing the method taxonomy diagram. The class name references the CIDS of the class in which the routine was defined and which also contains a list of all the type diagrams and inheritance diagrams in which the class appears. The combination of routine and class names allows one to reference the protocol diagram and client diagram that apply to the method set. The



Contract Data Sheet:
(Routine-Class Pair)

Method Set Name:
(Name)

Method Taxonomy Diagram:
(Routine Name)

Owner:
Name MI Surname

Date Approved:

Description / Definition

Constraints

Figure 4-12
Method-set Contract

input classes, output description, and the process information may be included as constraints. The description/definition component in the contract will be a textual description of the method contract and may include the method-set rationale.

In a design, the method sets are grouped together by related contracts to form a method taxonomy for a particular type of system behavior or routine.

These groupings represent the contract framework from the least-specific to the most-specific contract for a specified type of behavior. The method taxonomy diagrams provide a graphical means for illustrating how groups of methods are related in the design.

4.2.2 Method Taxonomy Diagrams Symbol Set

The two symbols used in a method taxonomy diagram are boxes and arrows. The boxes represent method sets and the arrows denote an additional constraint relationship or a redefinition relationship. The arrows in the diagram point from the less-specific to the more-specific method set. A method taxonomy diagram may be laid out either left to right or top to bottom on a page from the most-general to the most-specific method set.

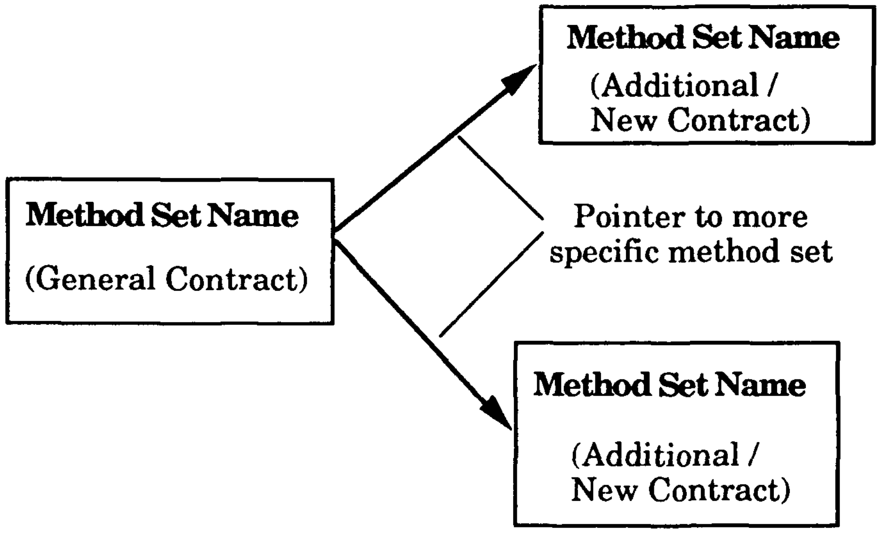

Method Set Name
(Additional / New Contract)
Method Set Name
(General Contract)
Pointer to more specific method set
Method Set Name
(Additional / New Contract)

Figure 4-13
Method Taxonomy Diagram Symbols

In Figure 4-13, the method set on the left represents the set of methods that satisfy a less-specific contract. These additional contracts denoted by the arrow may be conflicting or nonconflicting with respect to the contracts in the immediately more general methods. This means that the contract for the method sets at the end of an arrow will have all the constraints placed on the preceding method sets plus some additional constraints which may in some manner override or shadow those that are inherited. In either case, the additional contract makes the method sets more specific. The kind of contract relationship—redefinition or additional contract—will be specified using a "+" or "&" auxiliary symbol, respectively, in the inheritance diagram in which the class feature is defined. If the contract is being shadowed or redefined, its attribute in the class-inheritance diagram will include a "+" prefix and the method-set contract must be stated in full if there is no way of defining which contract items in the more-general contract are being shadowed (i.e., the notion of adding to the constraints of a more general method set will not apply).

4.2.3 Understanding a Method Taxonomy Diagram

The method taxonomy diagrams of IDEF4 are a way of organizing the contract structure for a class of behavior. They provide a means for graphically illustrating the additional or new contracts that are provided for a routine. By convention, the name of a method taxonomy diagram is the routine that is being described.

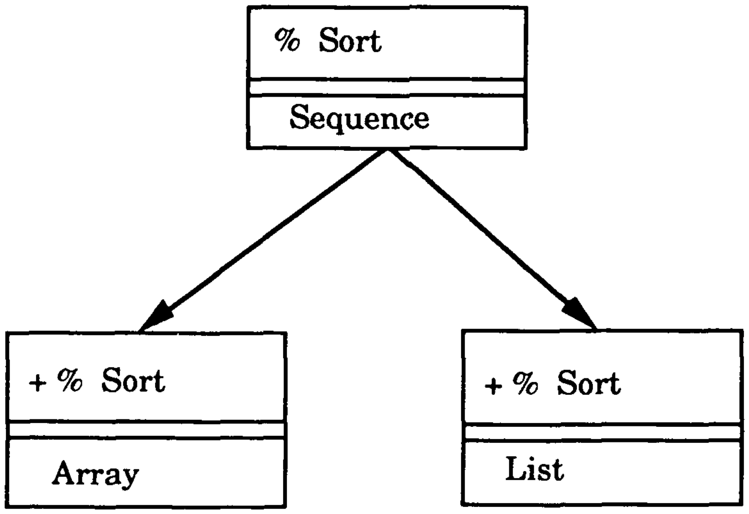


Figure 4-14
Class Structure with Additional Contract

Figure 4-14 illustrates a partial class hierarchy. In this hierarchy, a *Sort* routine is defined in the root class *Sequence* and inherited by both subclasses *Array* and *List*. In each of these classes, the routine has a new contract (e.g., additional and/or new constraints). This is illustrated by the *Sort* method taxonomy diagram as shown in Figure 4-15.

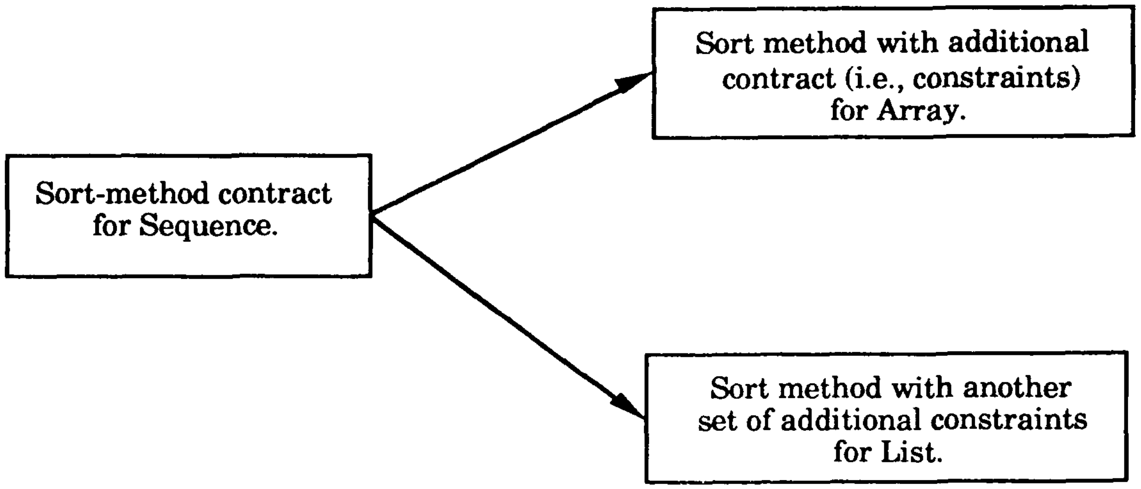

Sort-method contract for Sequence.
Sort method with additional contract (i.e., constraints) for Array.
Sort method with another set of additional constraints for List.

Figure 4-15
Layout for Sort Method Sets in Sequence Structure

It is important to note that every method taxonomy diagram is associated with a generic activity. In Figure 4-15, the generic activity is *sorting*. Every method set in the diagram will relate to the operation of sorting or to a sort contract for some routine-class pair. For readability, it is desirable to name the method sets as a concatenation of 1) an activity qualifier; 2) the operation, procedure, or function that the methods in the set would perform; and 3) the class name. The reason for the activity qualifier is to enable one to distinguish between method sets that have the same routine-class pair. An example of this situation would occur if the designer defined more than one sort routine (e.g., a bubble sort, insertion sort, and a heap sort) in the class *array* and named each one "sort." In this case, the sort m :hod sets would be named heap-sort-array, insertion-sort-array, and bubble-sort-array. The feature definition in the class-inheritance diagram would need an explicit dispatch containing the full name of the method set to which it is being dispatched.

The key to understanding a method taxonomy diagram is to understand the contracts associated with the method sets in the diagram. The boxes in a method taxonomy diagram represent method sets. Method sets are connected by arrows that point from a more general method set to more specific ones. Thus, we can say that the arrows in a method taxonomy diagram denote an additional contract. The additional contract represented in the diagram may provide constraints that are either conflicting or nonconflicting (see Figure 4-16).

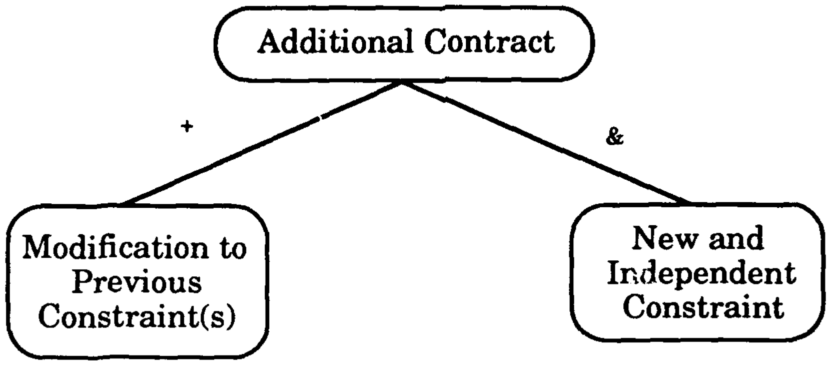

Additional Contract
+
&
Modification to Previous Constraint(s)
New and Independent Constraint

Figure 4-16
Additional Contract

Conflicting constraints are those that specialize or redefine the contract of a previous method set in some manner. In a CLOS program, this is analogous to the definition of a new primary method that shadows any less-specific methods. Nonconflicting constraints merely represent the addition of constraints that do not shadow or are not affected by any of the so-called inherited constraints. The nonconflicting constraints provide for the "pure" notion of "additional contract" and are, as a rule, sets of pre- or post-conditions that are expected to apply to a method in the primary method set. The concept of pre- and post-conditions implies that methods in the New method set would execute a primary method either preceded or followed by some additional method. In the CLOS model, this behavior is accomplished by before, after, around, and primary methods with an appropriate method-combination strategy. With C++, one would explicitly call the before and after methods to perform these operations or add the before and after method capability to C++. Both techniques are well documented in C++ reference books.

The important point to note when designing or reading a method taxonomy diagram is to first identify the behavior class of the diagram by which it is named. This will provides the reader with a general understanding of what is being illustrated and what is to be implemented. However, to fully understand the relationships between the method sets in the diagram, it is necessary to study the individual CDSs. For this reason, if the intent is to shadow some or all the constraints in the more general method set, then the CDS must contain a full and complete listing of the New method-set constraints. Furthermore, a CDS should state whether the contracts supersedes all preceding contracts or adds pre- and/or post-(nonconflicting) conditions. The layout of the diagram itself will provide only a general notion of an ordering from less specific method sets to more specific method sets. The CDSs provide the actual relationships between the method sets.

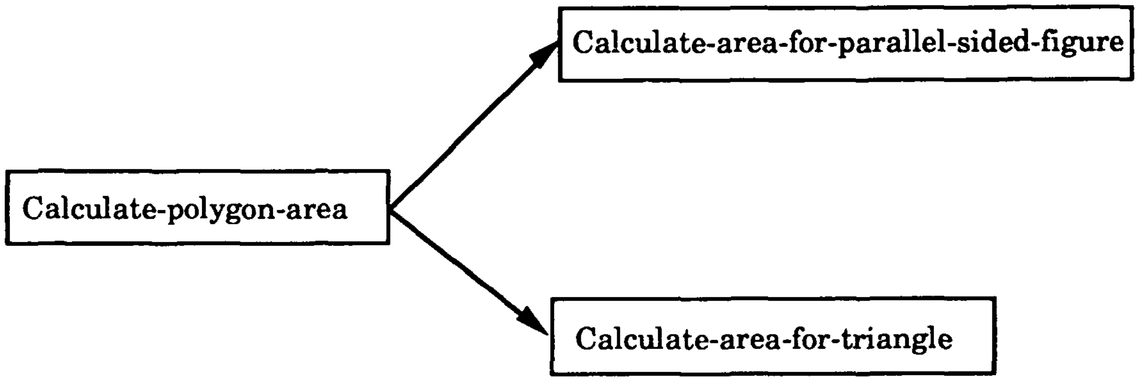

Calculate-polygon-area
Calculate-area-for-parallel-sided-figure
Calculate-area-for-triangle

Figure 4-17
Area Method Taxonomy Diagram

Figure 4-17 illustrates a method taxonomy diagram for *Area* calculation with additional, conflicting constraints. In this diagram, the constraint on the method set *Calculate-polygon-area* could indicate that methods in this method set will calculate the area of any object classified as a *polygon*. However, both of the other method sets in the diagram represent specializations of the constraints placed on the first. ***Calculate-area-for-triangle*** will calculate the area only for ***polygons*** that are triangles; the other method set represents those methods that will calculate the area only for figures with parallel sides. In both cases, the new or additional constraint supersedes or specializes the contract on the method set ***Calculate-polygon-area***, requiring a more restrictive or specialized type of behavior.

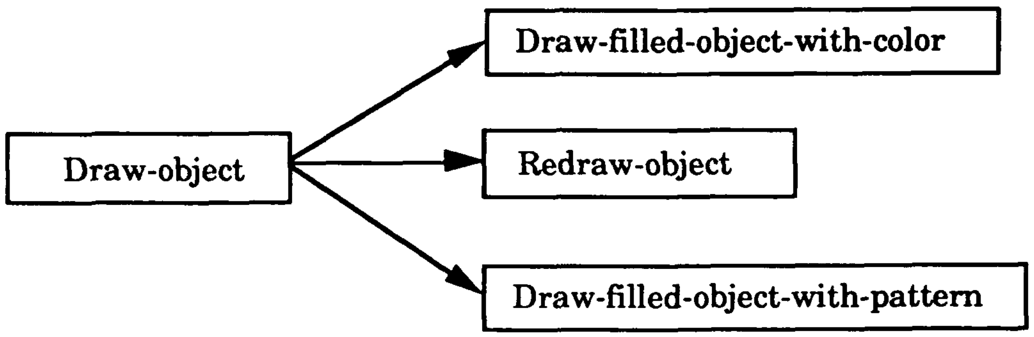

Draw-object
Draw-filled-object-with-color
Redraw-object
Draw-filled-object-with-pattern

Figure 4-18
Pure Additional Contract

Figure 4-18 illustrates a *Draw* method taxonomy diagram. The figure provides some insight as to the notion that the arrows in the diagram represent additional contracts. Without examining the CDSs themselves, one cannot be positive about the interpretation of the diagram. The following would be found in the CDSs associated with the method sets.

Draw-object is a primary method which draws an object.

Draw-filled-object-with-color the post-condition, "Color the object after it has been drawn," to the draw-object contract.

Redraw-object has the constraint (pre-condition) that "before drawing the new object the old object must be erased."

Draw-filled-object-with-pattern has the contract (post-condition) that "after drawing the object it is to be filled with a pattern."

The CDSs reveal that the most specific method sets expect the more general method set to execute either before or after they execute.

Method taxonomy diagrams have another application that is somewhat removed from the design of an individual system. This application is as a means of classifying and organizing method sets that perform operations common across a wide variety of systems. They can provide a catalog of previously coded methods. If a particular contract is very widely used and studied (e.g., for sorting), the corresponding method set and its subsets may form quite a complex taxonomy (see Figure 4-19). Such a taxonomy may serve as a resource for the designer. Taxonomies of this level of complexity also illustrate the relative independence of a method taxonomy from routine class-inheritance.

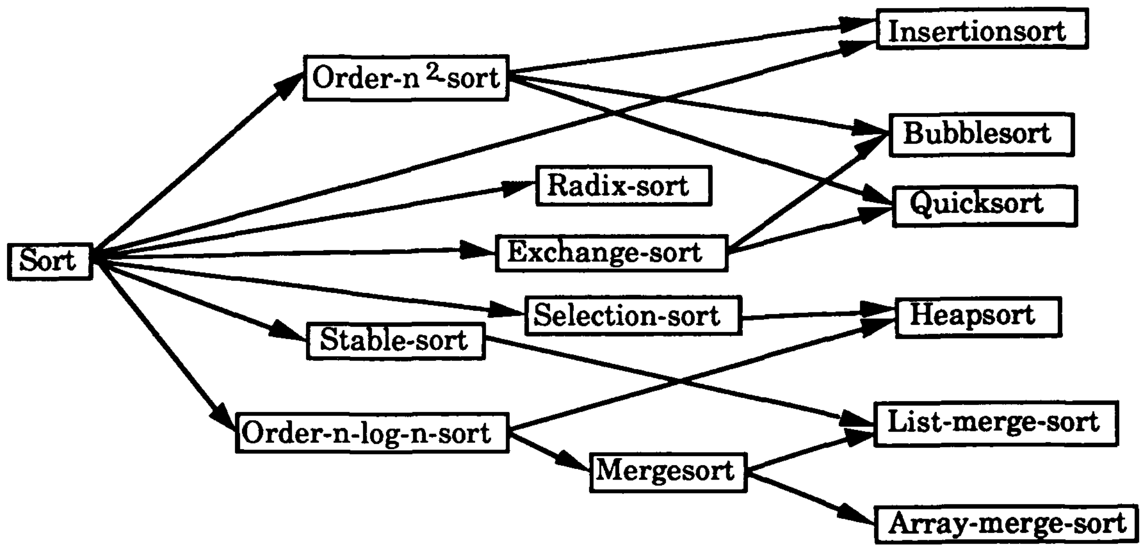
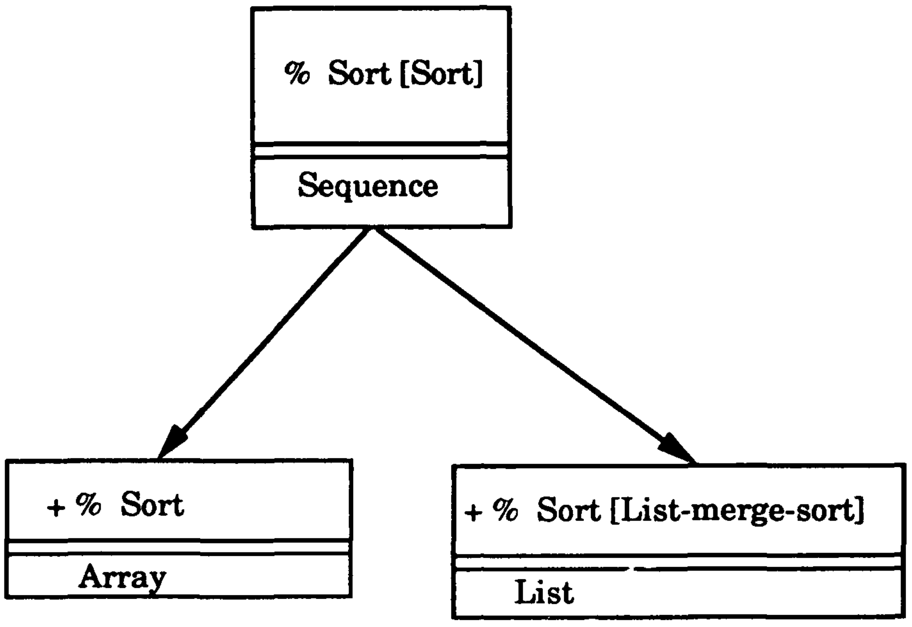


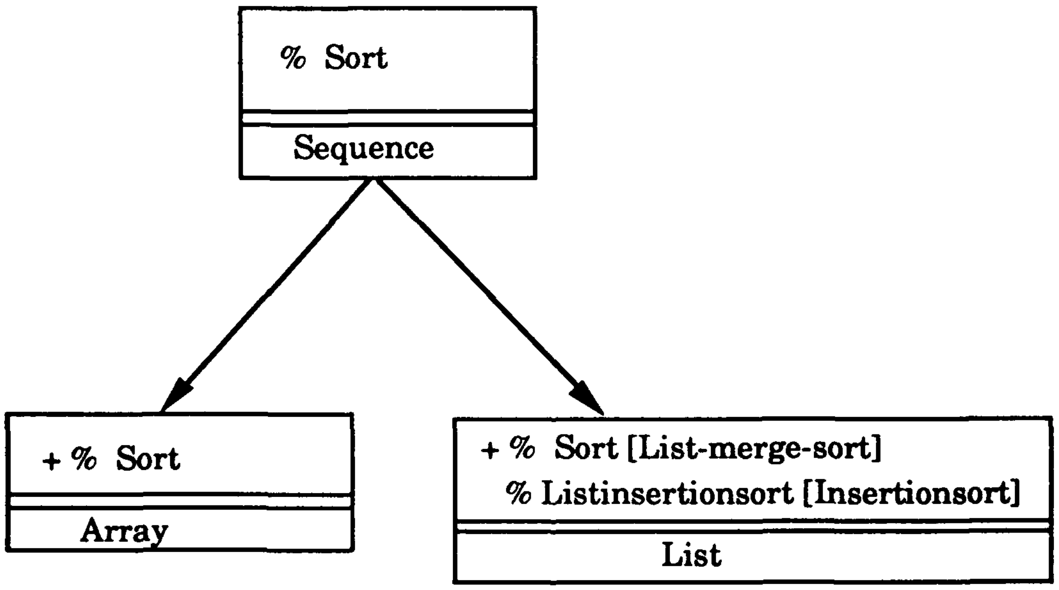
Figure 4-19
Sort Method Taxonomy Diagram

Figures 4-19 and 4-20 illustrate how method taxonomy diagrams are referenced from other components of the design using explicit dispatch mappings. Dispatch mappings must be explicitly defined when a routine-class pair picks out more than one method set.

Figure 4-20(a) illustrates that in the class *List*, the routine *Sort* is redefined and will be dispatched to the method set *List-merge-sort.* The term "dispatched" refers to the way of indicating which method-set contract is associated with the class-routine pair (see Section 4.6.) A comparison of



(a)


% Sort
Sequence
+ % Sort
Array
+ % Sort [List-merge-sort]
% Listinsertionsort [Insertionsort]
List

(b)

Figure 4-20
References to Method Taxonomy Diagram

Figure 4-20(b) and Figure 4-19 infers that the sort method taxonomy diagram does not necessarily group the method sets in the same hierarchy that the routines are grouped by a class-inheritance graph. Along with the redefined sort routine, another routine, *Listinsertionsort*, is defined in *List*. This routine is dispatched to *Insertionsort*. The ordering of the method sets in the diagram of Figure 4-20(b) is not the same as that in the sort method taxonomy diagram of Figure 4-19. Both *Sort* and *Listinsertionsort* are legitimate routines, and both are dispatched (see Section 4.6) to method sets in the same method taxonomy diagram.

Method taxonomy diagrams are also an important aid for the designer in identifying opportunities for reuse of functionality in the design. Strong indicators for the reuse of functionality are 1) the degree of fan-out in the method taxonomy diagram, 2) the number of levels in the method taxonomy diagram, and 3) the degree of similarity in the CDSs of methods. Method taxonomy diagrams that exhibit a high degree of fan-out are a strong indication that the introduction of more abstract classes may allow one to increase the degree of reuse in the design. This is illustrated by the diagram shown in Figure 4-21.

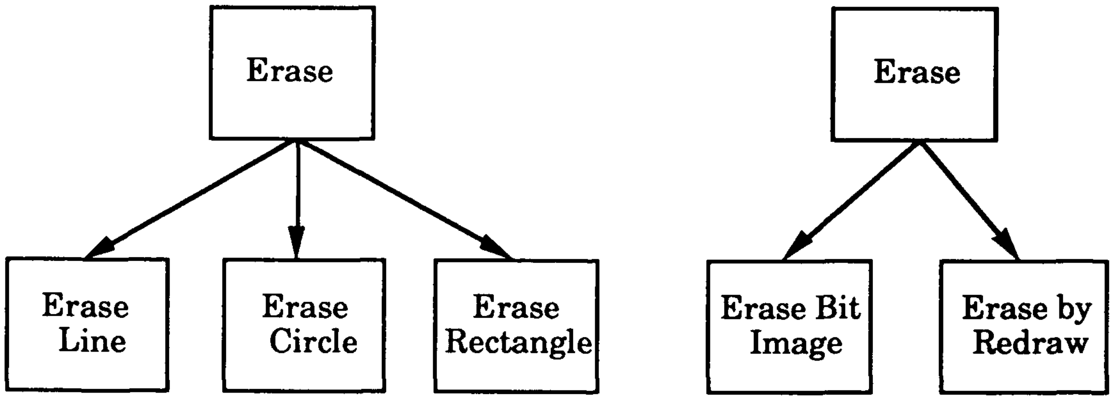


Figure 4-21
Polymorphism Evaluation with Method Taxonomy Diagrams

The method taxonomy diagram on the left side of Figure 4-21 would be considered less desirable than the one on the right. This is because the diagram on the left requires the implementation of three different sets of methods for a simple graphical object erase. The design on the right, by contrast, is more generic (i.e., it is based on a more abstract class). The resolution of the erase routine is transferred to the draw method taxonomy diagram which simply applies a *draw* function to the object using an inverse color mapping. Considerably fewer method sets need be designed as new graphic classes are introduced. For example, the addition of a new class in the system described in Figure 4-21 would, worst case, require the definition of a draw routine for the new class.

4.3 Type Diagrams

Initial exposure to object-oriented programming may lead one to believe that the primary relations between object classes are those defined in the inheritance lattice. However, many of the interesting relations, from an application execution point of view, are established implicitly through the values of the attribute features of the object classes. One of the primary differences, noted by applications and database designers switching from the functional and relational paradigms to the object-oriented paradigm, is the lack of emphasis on the definition and management of keys and referential-integrity problems. Because objects in an OOPL are uniquely identified by their system-defined handle, access is generally structured around roles that objects participate in relative to other object classes. These roles are often established by the object class attributes. The design of these roles is generally for the purpose of circumscribing a particular behavior. Thus, the related object type is often a key constraint on the appropriateness of an object assuming a particular role. Management of this role-definition process is accomplished through IDEF4 type diagrams. In the IDEF4 discipline, all attributes have a return type, defined as the type of their return value. The type diagrams in IDEF4, a part of the class submodel, provide graphical and textual notations to display the return types of the attributes of the classes in the IDEF4 model.

4.3.1 Type Diagram Symbol Set

The purpose of type diagrams is to provide a visualization of the return type for the class features that return a value. For this reason, the type diagrams employ the class box of the inheritance diagram as shown in Figure 4-22. However, in a type diagram, only features specified to be attributes, functions, or slots are shown. Auxiliary feature symbols (&, +, !, ^, and *) and the export line are not used.

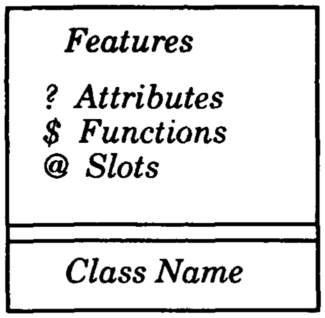


Figure 4-22
Class Box in a Type Diagram

A type diagram contains class boxes and type links. The type link serves two purposes:

- highlight return types, and
- show inverses or partial inverses.

The type links between class boxes will always have at least one end that extends into the class box, pointing toward one of the features in the class box. This indicates that the feature adjacent to the type link that penetrates the class box, returns values of the type of elements the class box on the opposite end of the type link.

Links Without Inverses or Partial Inverses

1. The return type for a feature in one class may be an instance of another class as shown in Figure 4-23. This is indicated with a line that extends from the feature name within the class box, through a dot on the outside of the

box, and to a dot on the outside of the class box of the return type. For example, the diagram in Figure 4-23 illustrates that the return type of feature *f* in an instance of class *A* is an instance of class *B*.

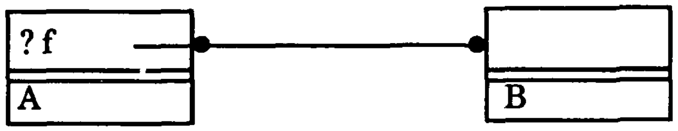


Figure 4-23
Return Type is an Instance of a Class

2. The return type for a feature may be composed of a structured collection of a type. The link used to indicate types that are structured collections is similar to the simple type link in Figure 4-23, except that a "half diamond" or *crow's foot* (>) is used on one end of the type link. Figure 4-24 illustrates that the feature *f* of class *A* returns values of some type that is constructed from class *B*. For example, this symbol would be used if *f* represented a list of objects of type *B*. A CDS or CIDS would be used to supply the exact type of structure represented by *f*.

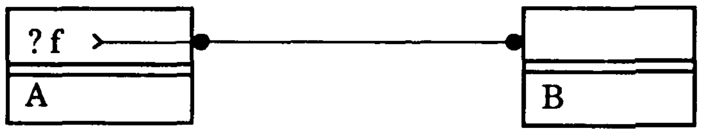
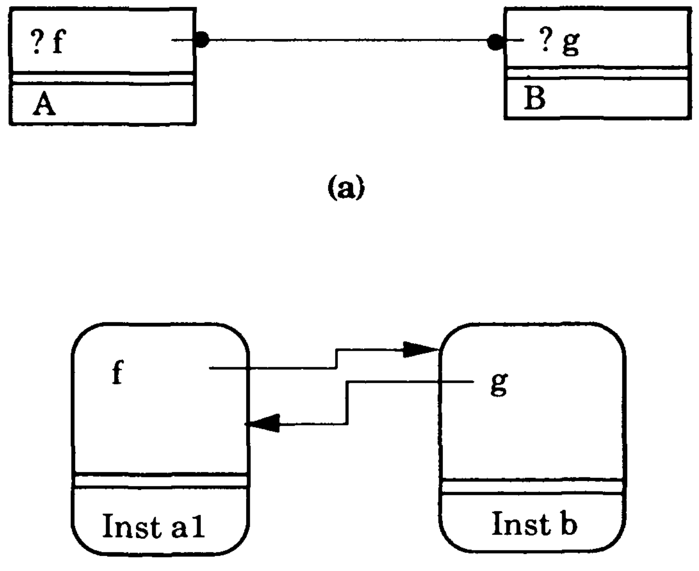


Figure 4-24
Return Type Constructed from Some Class

In the design of object-oriented systems that support persistence (i e., those intended for ODBMS implementation), the decision relative to inverses and partial inverses becomes important. The concepts of "inverses" and "partial inverses" warrant illustration. In Figure 4-23, an instance of type *A* will have a feature *f* that will return a value that is an instance of type *B*. However, an instance of type *B* does not carry any information about which (if any) instances of *A* it is associated with. This lack of an inverse function associated with type *B* is indicated by the termination of the relation link at the edge of the *B* class box. The same reasoning would apply to illustrations of the type displayed in Figure 4-24, no inverse function is indicated.

Links With Inverses or Partial Inverses

1. Figure 4-25(a) illustrates how an inverse relationship between instances of two types is displayed in a type diagram. The diagram illustrates that there is a feature *g* in type *B* connected to the link between *A* and *B*. This indicates that the feature *g* of an instance of type *B* contains a value that is an instance of type *A* and it is, in fact, just that instance of *A* that has that instance of type *B* as its *f* feature. Thus, we can think of the instances of type *B* as having "where used" pointers to instances of type *A*. Since there is no *crow's foot* on the end of this link, we know this relationship to be functional in both directions; hence, we refer to *f* of *A* as having an inverse *g* of *B*. Similarly, we can refer to *g* of *B* as having *f* of *A* as its inverse.

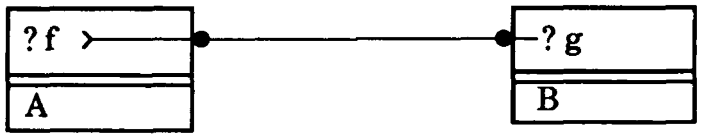


(b)

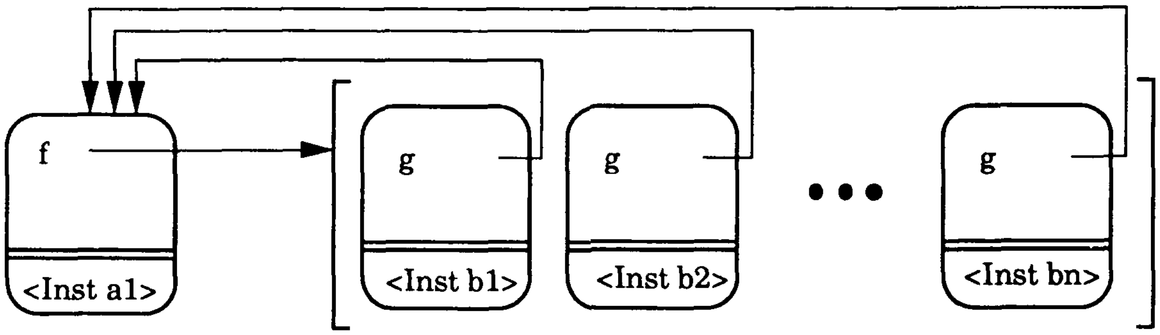
Figure 4-25
Inverse Relationship

2. Partial inverses are an issue when considering relations other than one-to-one between types (see Figure 4-26). Figure 4-26(a) illustrates that for an instance of *A*, the feature *f* returns values of some type constructed from instances of type *B* and the feature *g* of any of those instances of *B* will return that instance of *A*. If we have an instance *a* of type *A*, then *f*(*a*) is some structure of

objects constructed from instances of type *B*, and for any *b* of type *B* that is an element used in constructing *f(a)*, *g(b)=a*.



(a)



(b)

Figure 4-26
Partial Inverses

In a type diagram, every feature return type must be shown, but the designer is free to use either a textual notation, a graphical link, or both.

Concatenating the attribute name followed by the feature return type reduces the complexity of a diagram (see Figure 4-27). This alternative syntax is used in larger diagrams to reduce the unnecessary clutter of the diagram by eliminating a number of links. It can also be used by the designer to de-emphasize certain relations and focus the attention of the design reviewers on specific relations (i.e., those shown with a link). Finally, this approach is often used for the more common data types such as integer and Boolean. With these data types, the class boxes can be omitted from the diagram, along with all graphical type links to them, thus simplifying the diagram and avoiding unnecessary clutter.

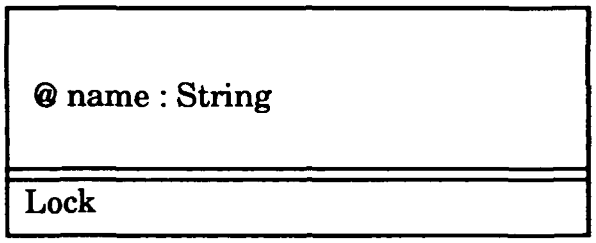


Figure 4-27
Return Type Displayed Textually

4.3.2 Understanding Type Diagrams

Now that we have discussed the symbols used to construct a type diagram, we will look at how these symbols are combined to form a completed diagram. From a design management point of view, the type diagrams provide a visual representation of the relations between classes implicit in the definitions of the types of each feature. This information is not visible in the inheritance lattice. Experience has shown, however, that management of these interconnections is critical to the development of large object-oriented systems. Through the use of the type diagrams, the programming team can communicate intended relations. They can also use the type diagrams to quickly review the impact of a proposed change.

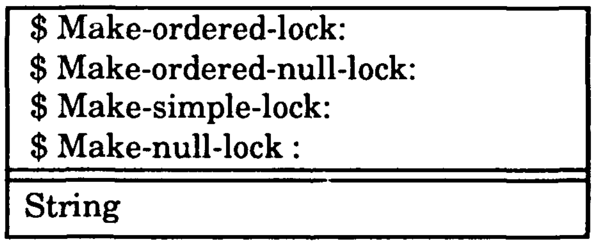


Figure 4-28
Class Box *String*

In Section 4.1.2, Figure 4-7 illustrated the class-inheritance diagrams using a Network Manager System. We will continue with that theme in this section on type diagrams. From that class-inheritance diagram, we take the class *String* (see Figure 4-28). Each feature in this class is a function that has a *return type* which is an instance of some other class.

Figure 4-29 indicates that the function *make-ordered-lock* returns an object which is of type *ordered-lock*. The function *Make-ordered-null-lock* returns an object which is of type *Ordered-null-lock*.

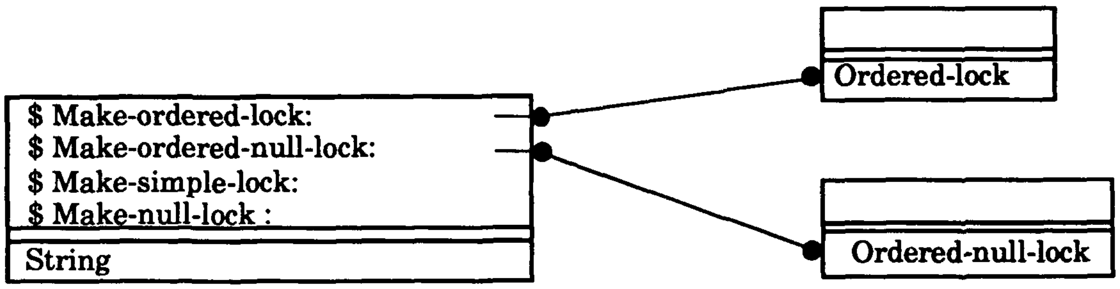


Figure 4-29
Simple Return Types

To show that the slot *Requests* is expected to contain a collection of objects of type *Print-request*, the link would be drawn with the half diamond in the class box by the feature *Request* (see Figure 4-30). Either the CIDS or CDS must provide the information that the *Print-request-queue* may contain the specifications for the type of structure. Figure 4-31 displays the completed type diagram.

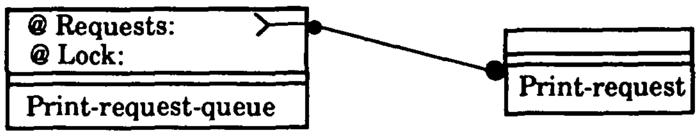


Figure 4-30
Return Type Constructed of Other Types

The situation may arise in which a routine may return one of two types. For example, in Figure 4-32, the type diagram can show that the feature *Color* returns an instance of the type *Color*. It is not possible to show that the feature *Color* will either return the color of the triangle if it has a color or *NIL* if there is no color. In other words, the graphical syntax of IDEF4 type diagrams (and protocol diagrams as well) will not allow the illustration of multiple return types for the return value of a class feature. If this situation occurs, the introduction of a more abstract class based on the different return types is necessary. If it is not possible to define a more abstract class, or if the extra types are only needed in exceptional conditions, then the situation should be stated as a constraint in the CDS or CIDS.

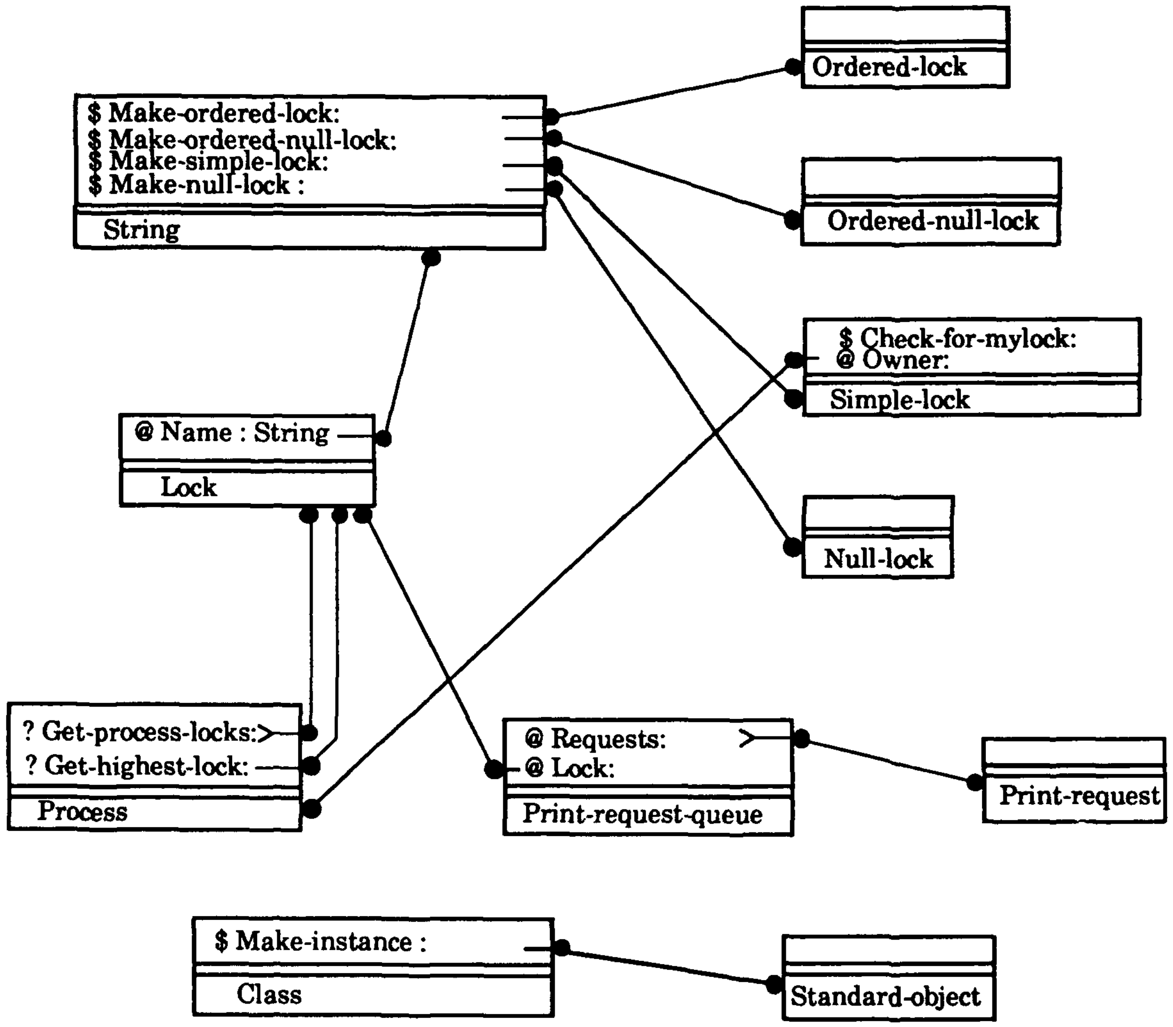
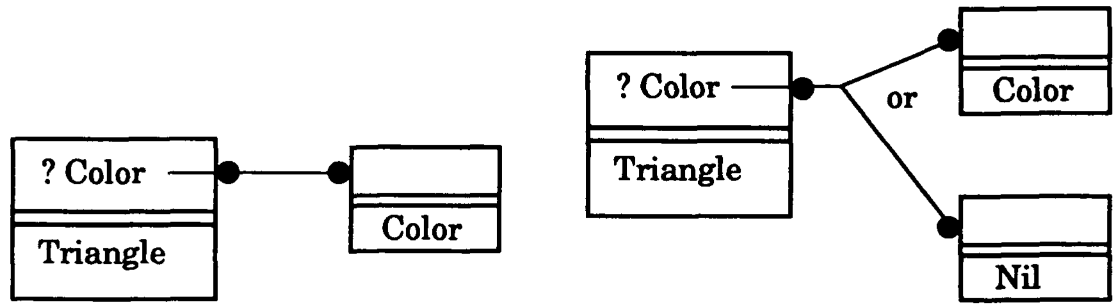


Figure 4-31
Type Diagram (Network Manager System)

4.4 Client Diagrams

In this section, we will cover the client diagram component of an IDEF4 model. Client diagrams, which are part of the method submodel, are used for algorithmic decomposition. They are the only IDEF4 diagrams that specify, however abstractly, the internal structure of methods.



(a) Allowable (b) Not Allowable

Figure 4-32
Allowable/Not Allowable Type Diagrams

4.4.1 Client Diagram Symbol Set

Client diagrams employ two symbols--boxes and links. The client diagram boxes represent features or feature-class pairs. The capitalized class name appears first, followed by a colon; the feature name appears in lower-case letters below it. The links that join these feature class boxes are referred to as client links.

These client links are shown as arrows with double barbs at both ends which point in the same direction (see Figure 4-33). The arrows point from the *supplier* (the feature that is called or referenced) to the *client* (the feature that *calls* or "makes a control reference"). The boxes themselves represent features or feature-class pairs. Client diagrams are typically small and center on a single focus routine (strictly a routine-class pair).

4.4.2 Understanding a Client Diagram

In a client diagram, the links between boxes represent control references or the existence of programming-language "subroutine calls" from one routine to another. Client diagrams specify the intended algorithmic structure of a method.

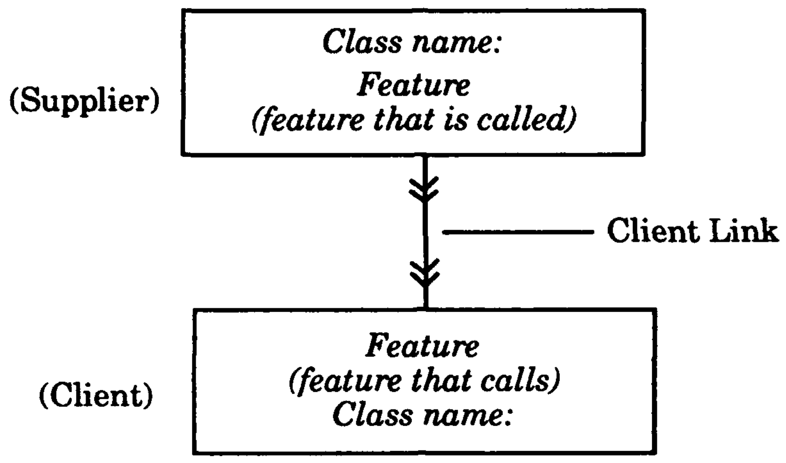


Figure 4-33
Client Diagram Symbols

To show what is intended by a client link, we will use the link from *Process:Process-wait* to *Simple-lock:seize* as an example (see Figure 4-34). For the routine-class pair *Simple-lock:seize* to appear as a *client* in a client diagram (i.e., at the front of a client link), *Simple-lock* must be a class in which *seize* is directly present (i.e., *seize* may not be virtual in *simple-lock).* Recall from Section 3.2.3 that directly present features are those whose names are displayed in the class box of a class. Directly present features are either defined in the class or redefined in the class in which their name appears. Therefore, the appearance of *Simple-lock:seize* as a client indicates that the designer intends to provide a new or redefined method for *seize* in *Simple-lock* (a method not provided in any *Simple-lock* superclasses). It also specifies that implementations of the method set for *seize* are intended to call the feature *Process-wait*, which is defined in the class *Process.*

Note that the diagram defines that implementations for *seize* on the class *simple-lock* will call the implemented routine *process-wait* directly, not some generic routine for *process-wait* (i.e., the diagram shows that the dispatching for this example may be performed statically). If the classes associated with the routines have not been specified in the diagram, the dispatching will occur at run time for any implementation. For an implementation in C++, this would indicate the need for virtual functions.

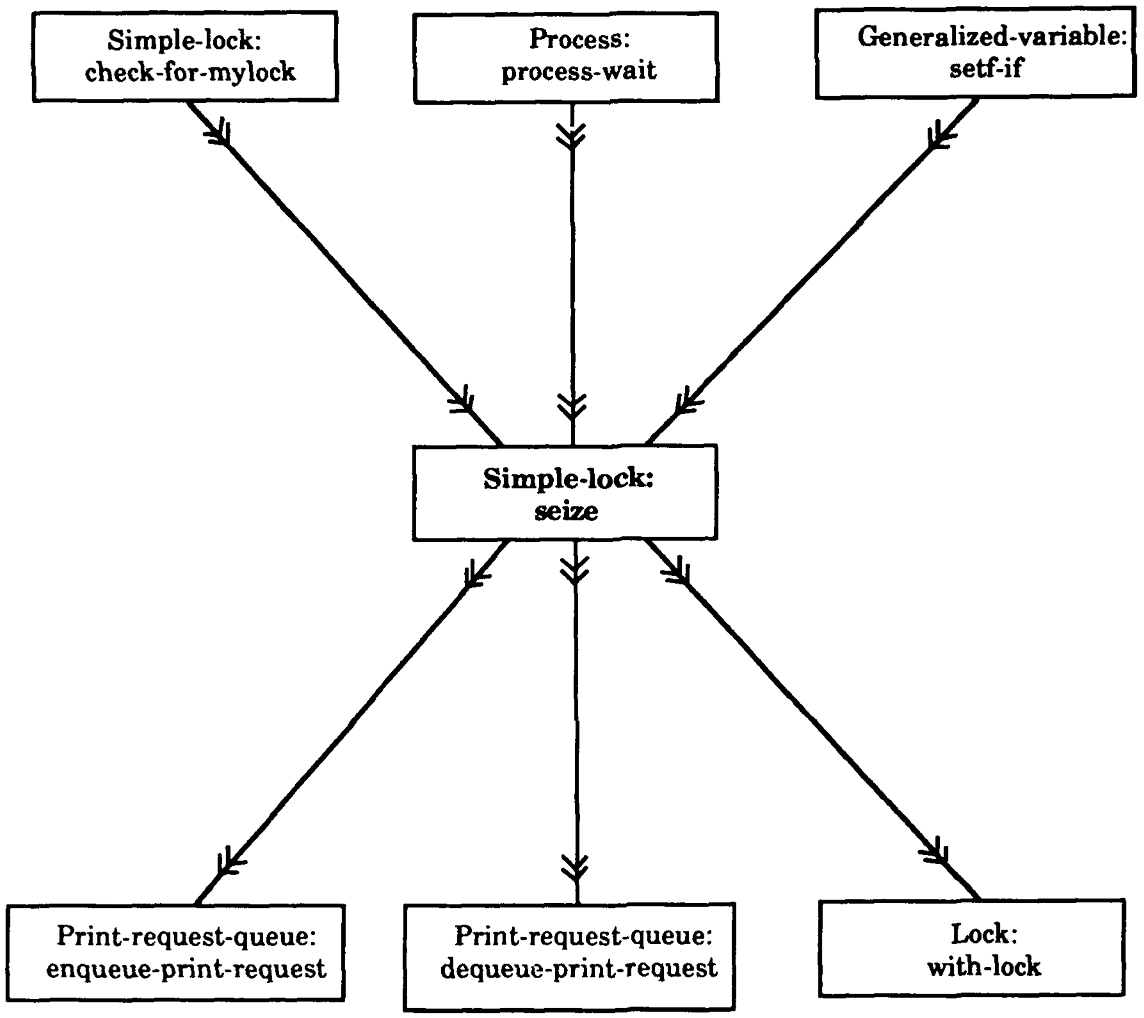


Figure 4-34
Client Diagram for *Simple-Lock Seize*

4.5 Protocol Diagrams

When a feature is activated, it is activated with respect to a particular object which is an instance of some class in which that feature is present. According to the terminology of some OOPLs (such as Smalltalk and Old Flavors), the feature is a *message* which is *sent to* the object. In other languages (such as New Flavors and CLOS), the feature is a *generic function* which is *called* with the object as an argument. IDEF4 adopts this *function* which is *called* with the object as an argument. IDEF4 adopts this latter view, except that the term *feature* rather than *generic function* is used. In IDEF4, a feature is considered a generic operator which is called on at least one argument (the self argument). This concept naturally leads to the consideration of the presence of other arguments. These other arguments are referred to as secondary argument places or secondary arguments. IDEF4 does not require a feature executed on an object to return a value (not all features are attributes).

The IDEF4 type diagrams provide the *return types* of the attributes but do not address the arguments to features or their types. The term *protocol* refers to the specification of the interface of a feature with its inputs and outputs. IDEF4 introduces *protocol diagrams* (part of the class submodel) to represent this information. These small diagrams focus on the inputs and outputs for the specified feature.

4.5.1 Protocol Diagram Symbol Set

The symbols used to create a protocol diagram are illustrated in Figure 4-35. A simple box is used to denote the feature that is the focus of a particular diagram. Round-cornered boxes denote the argument places of the focus feature, modified class boxes for argument types, and type links (see Figure 4-35).

The name of the focus feature appears in the center of the diagram in a plain box. The focus of each diagram is a routine-class pair. In a completed IDEF4 design model, each routine-class pair should have a protocol diagram. The routine-class pair is the identifier for the diagram and is used to order the diagrams within the documentation. Boxes with rounded corners represent inputs and outputs. Input arguments are placed above the focus feature box; output arguments appear below. The general convention is to indicate the self argument as the first argument on the left. To avoid unnecessary duplication of type links, only the root class (in the entire inheritance submodel) of the feature is shown. If there is more than one root class, all are shown with multiple type links connected to the self-argument place. The arguments are connected to the focus feature in the manner indicated in Figure 4-35. Argument types are connected to the arguments (input and output) by type links. As stated previously, IDEF4 does not allow more than one self argument. The argument types are displayed as class boxes with only the class name given.

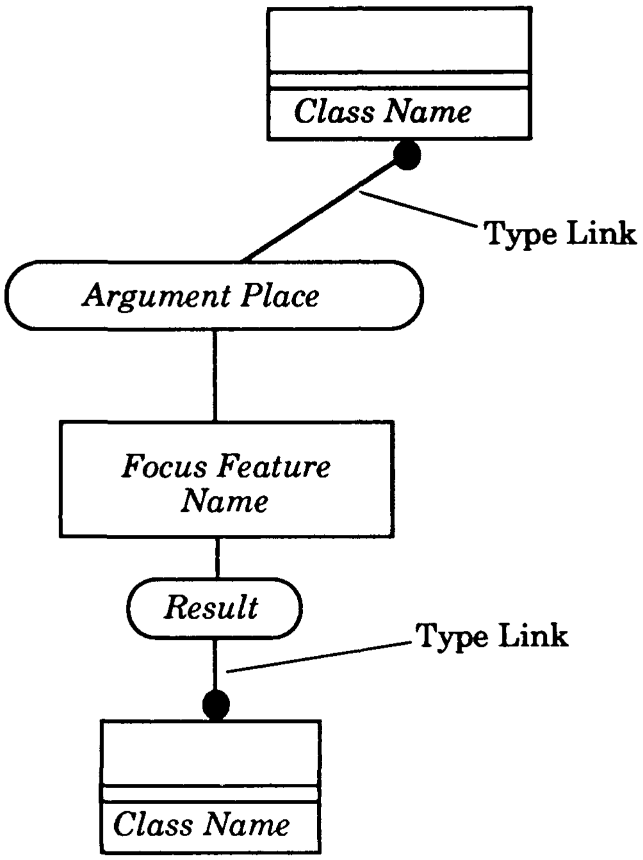


Figure 4-35
Protocol Diagram Symbol Set

4.5.2 Understanding a Protocol Diagram

Now that we have examined the protocol diagram symbols, we will look at how they are used to create a diagram. From the Network Manager System example, we have the class *Simple-lock* as shown in Figure 4-36 with the feature *Check-for-mylock*.

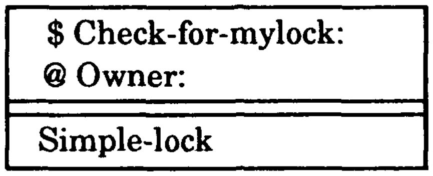


Figure 4-36
Class Box with *Check-for-mylock* Feature

The protocol diagram for the function *Check-for-mylock* as shown in Figure 4-37 would place this feature name in the focus feature box in the center of the diagram. The diagram shows the protocol for the routine-class pair *Check-for-mylock*: *Simple Lock*. In the diagram, the round-cornered boxes specify the argument names. The word "self" to the left of the argument name in the left-most argument box is a qualifier that indicates the argument is a primary argument. The remaining arguments to *Check-for-mylock*: *Simple Lock* appear to the right of this primary arguments. The primary argument in the diagram is *possibly-owned*; it is of type *Simple-Lock*. The one secondary argument, *possible-owner*, is of type *Process*.

A protocol diagram by itself provides a summary of the intended input and output characteristics of a feature. Protocol diagrams are also used to create view diagrams which show other information about the argument places and their types. Such view diagrams could not be created if protocol diagrams were not available as ingredients. A common type of view diagram could be created for the protocol diagram in Figure 4-37 by adding inheritance links for all the subclasses of *Simple-lock* in which *Check-for-mylock* is redefined (incidentally, there are none).

Usually, only routines are thought of as having more than one argument; however, but our discussion of protocols applies to all features, including slots. In fact, an attribute with only one argument and a return value can have a protocol, even though it is not yet known whether it is to be a function or a slot. The auxiliary feature symbol "!" (New taxonomic specification) can be used to allow the feature to be redefined more specifically as a function or a slot in a subclass, perhaps differently in different subclasses.

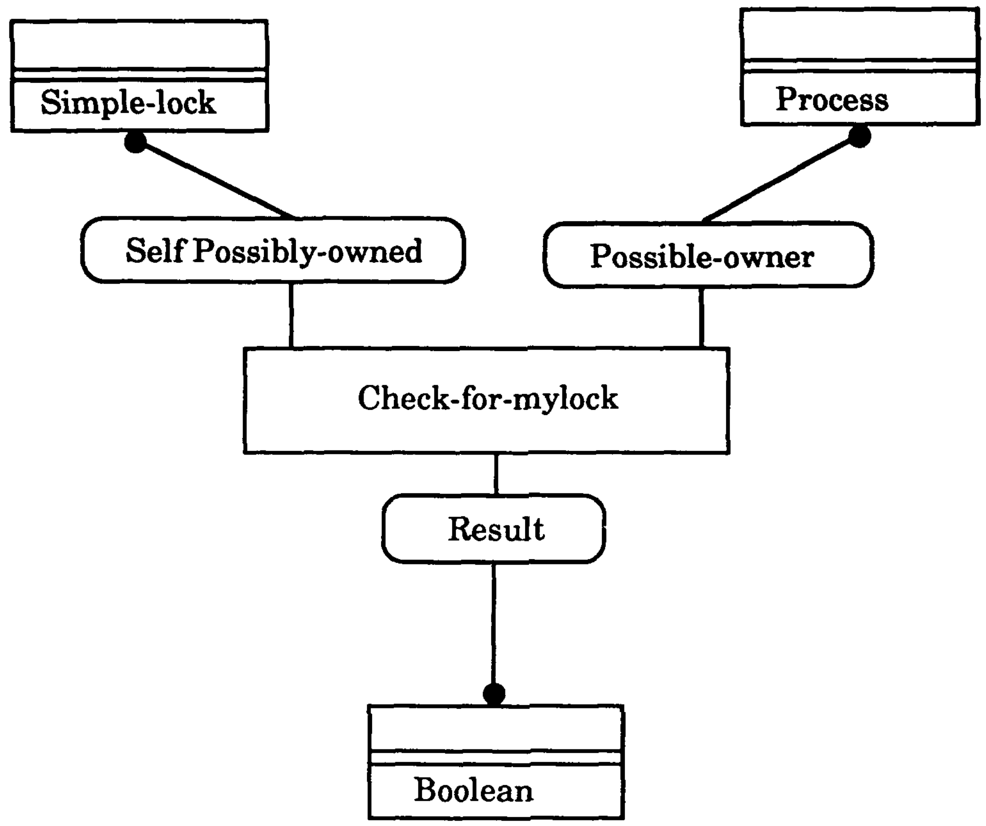


Figure 4-37
***Check-for-mylock*: *Simple-Lock* Protocol Diagram**

Normally, the protocol for a slot would only have the self argument and would return whatever value was stored in that slot in the instance to which it was applied. But it would be possible to imagine a slot that took additional arguments. For example, a slot could store not a single value but a database-like table of values. Such a slot, *f*, could be given a protocol as a multi-argument attribute in a class, *A*, without requiring the designer to commit himself as to *f*'s status as a slot (except perhaps in some subclass, *B*, of *A*). The additional arguments would be used as keys to find the right line of the table from which to extract the appropriate return value. The attribute *f* might be characterized as a function in another subclass, C, of A, not doing table lookup but calculating its return value. The distinction between slot and function need not be drawn in A at all.

4.6 Dispatch Mapping

Dispatch mapping, as interpreted in IDEF4, is similar to dispatching in an OOPL. In general, more than one method will be associated with a generic routine. In IDEF4, it is possible for a generic routine to have more than one method in the same class. In an OOPL implementation, only one method will be associated with each routine-class pair. In an OOPL, dispatching refers to the process of finding the appropriate method for a particular call to a generic function at compile time or run time. In IDEF4, a method set characterizes a set of methods which would serve as an implementation. IDEF4 adapts the term dispatch mapping or dispatching to refer to the association between a routine, a class, and a contract for a set of methods (or method set).

Recall that an IDEF4 model consists of a class submodel and a method submodel (see Figure 4-38). These two submodels are connected via the dispatch mapping. The dispatch mapping in an IDEF4 model is recorded either explicitly or implicitly in the inheritance diagrams and method taxonomy diagrams. When a diagram (inheritance or method taxonomy) includes dispatching, it indicates which method-set contract would be used to implement a particular routine-class pair.

Illustrating the dispatching using the two diagram types does not require changes to the general syntax of either diagram. Each diagram is drawn with the additional notation for the dispatching added. In a class-inheritance diagram, the dispatching is indicated by placing the method set name in brackets to the right of the routine name. In a method taxonomy diagram, the dispatching is shown by placing the routine-class pair in brackets below the name of the method set.

For example, consider the Network Manager System we have used throughout our diagram discussions. Figures 4-39 (a) and (b) illustrate a

Lock class-inheritance diagram and a *Seize* method taxonomy diagram, respectively. These diagrams can be used to show the mapping between the routines in the class-inheritance diagram and the method sets of a method taxonomy diagram. Figure 4-40 illustrates the class/routine/method-set mapping in an inheritance diagram for the *Seize* routine. In Figure 4-41, the same dispatching for the *Seize* routine is shown in the *Seize* method taxonomy diagram.

Dispatching is indicated similarly in a method taxonomy diagram and an inheritance diagram. However, because the focus of these diagrams is the method sets, the name of the routine-class pair is listed in brackets beneath the method set. Only the class in which the routine is directly present has been indicated.

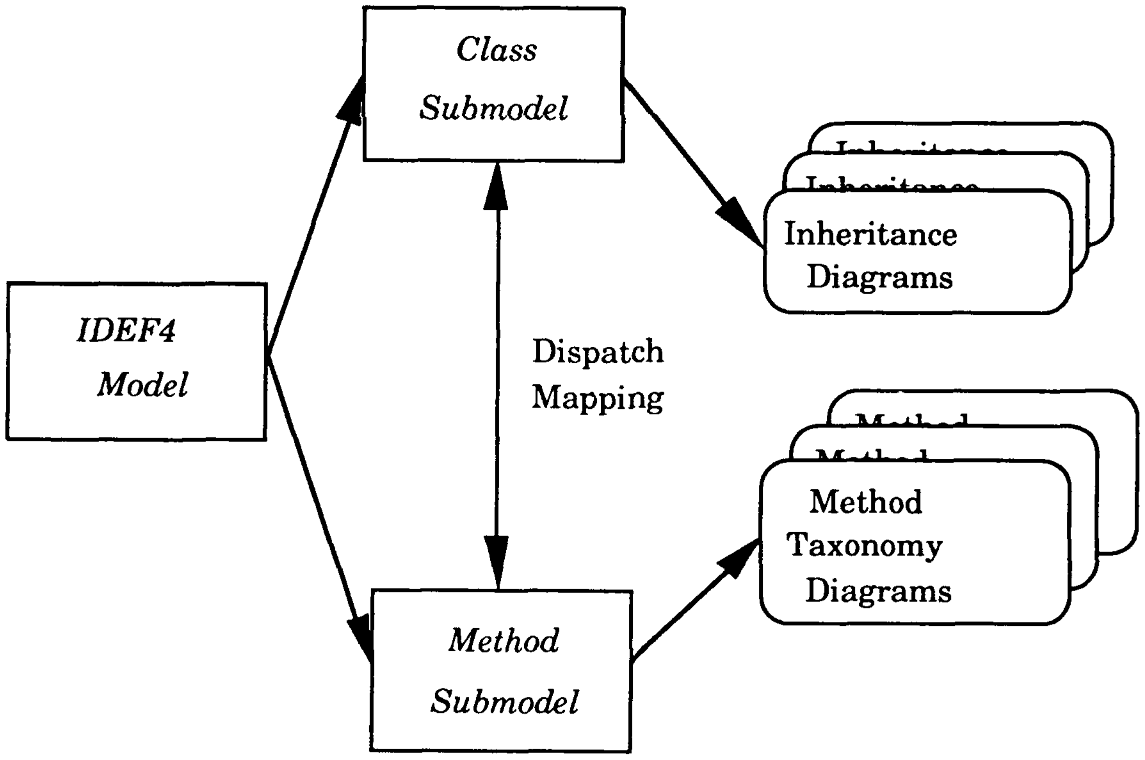
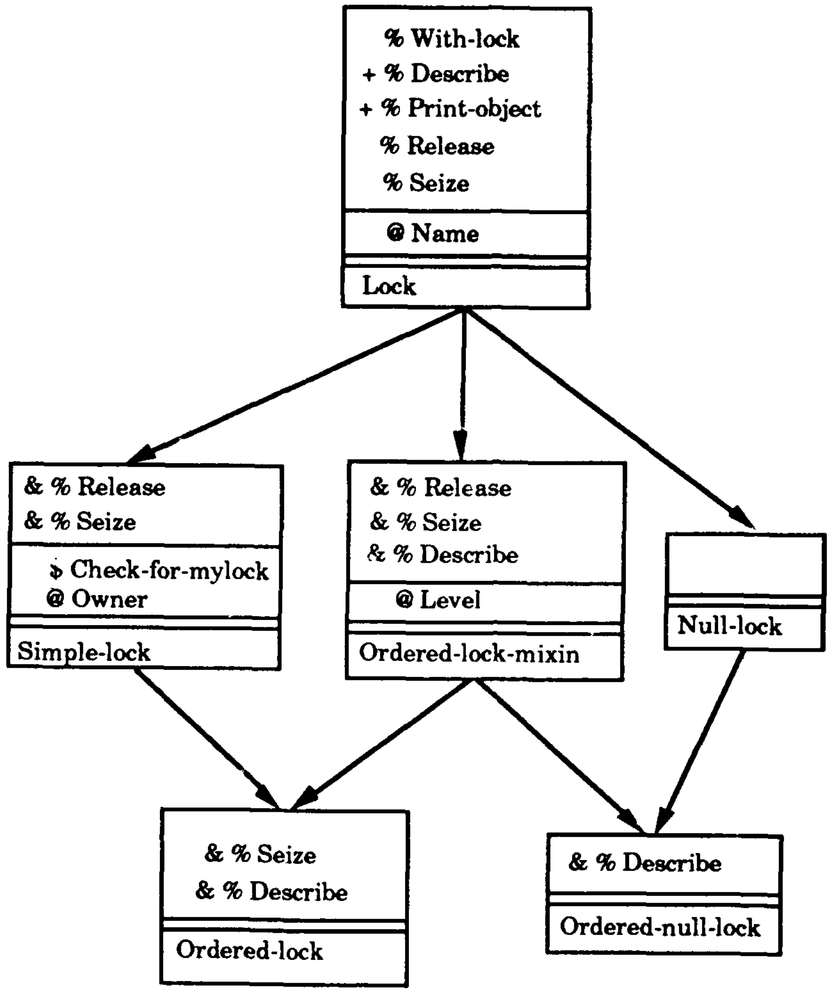
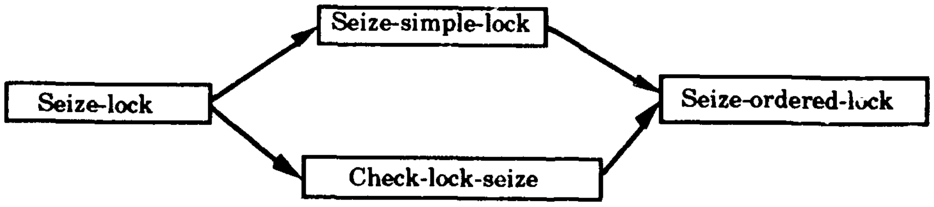


Figure 4-38
IDEF4 Model Components



(a) **Class-Inheritance**



(b) **Seize Method Taxonomy**

Figure 4-39
Partial Diagrams

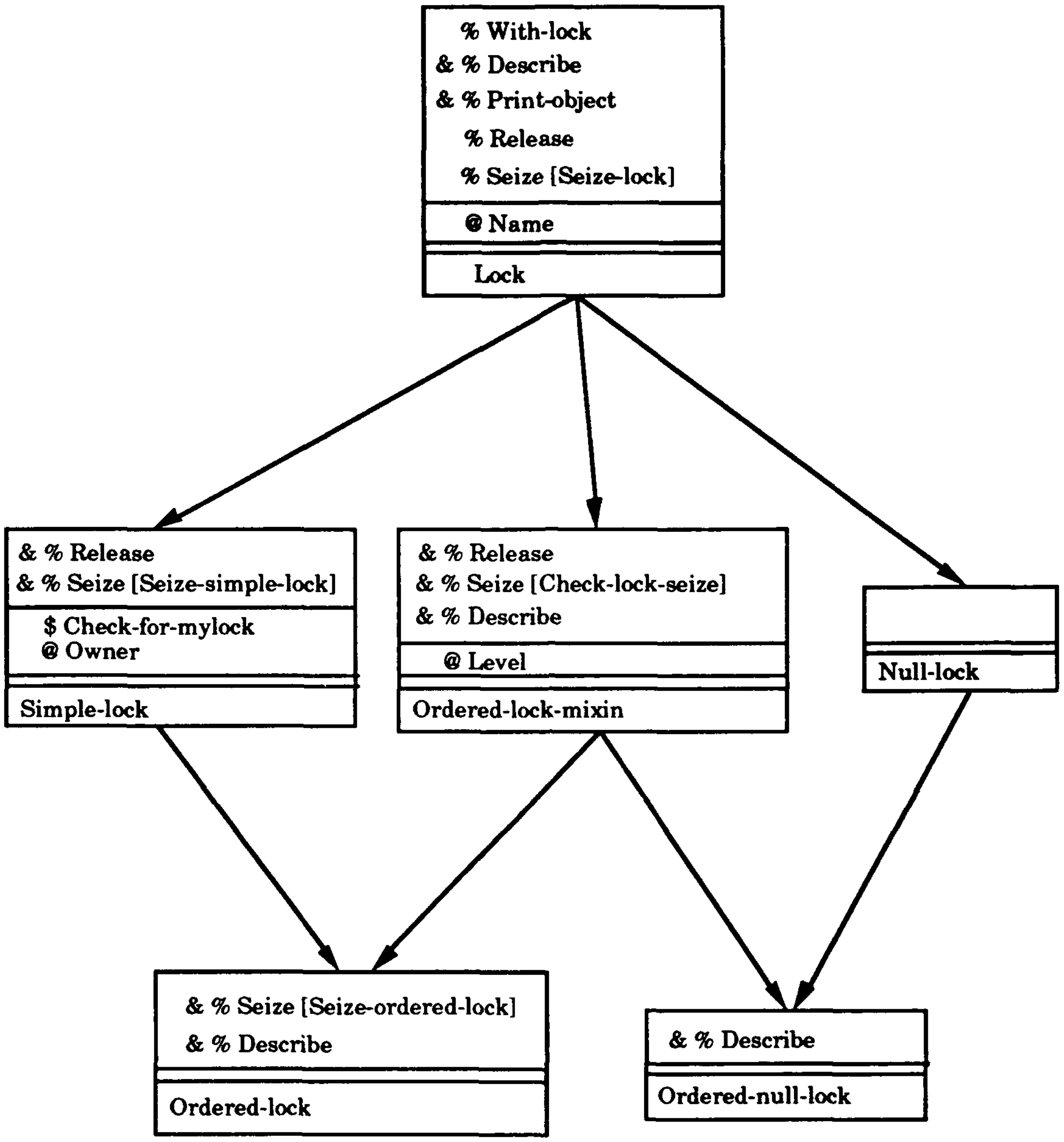


Figure 4-40
Inheritance Diagram with Seize Dispatch Mapping

4.7 IDEF4 Instantiation Language

The purpose of an instantiation language is to enable the development of test case scenarios. Test case scenarios, in turn, are used to validate the design and documentation of examples of intended design configurations.

Ultimately, this validation process aids the programmer in implementing the design. The graphical projection of the instantiation language looks much like an IDEF4 type diagram. The instantiation language uses a round-cornered box to represent instances of a class which is analogous to the IDEF4 class box. The value links used in the instantiation language are similar to the IDEF4 type links, but without the filled circles on either side.

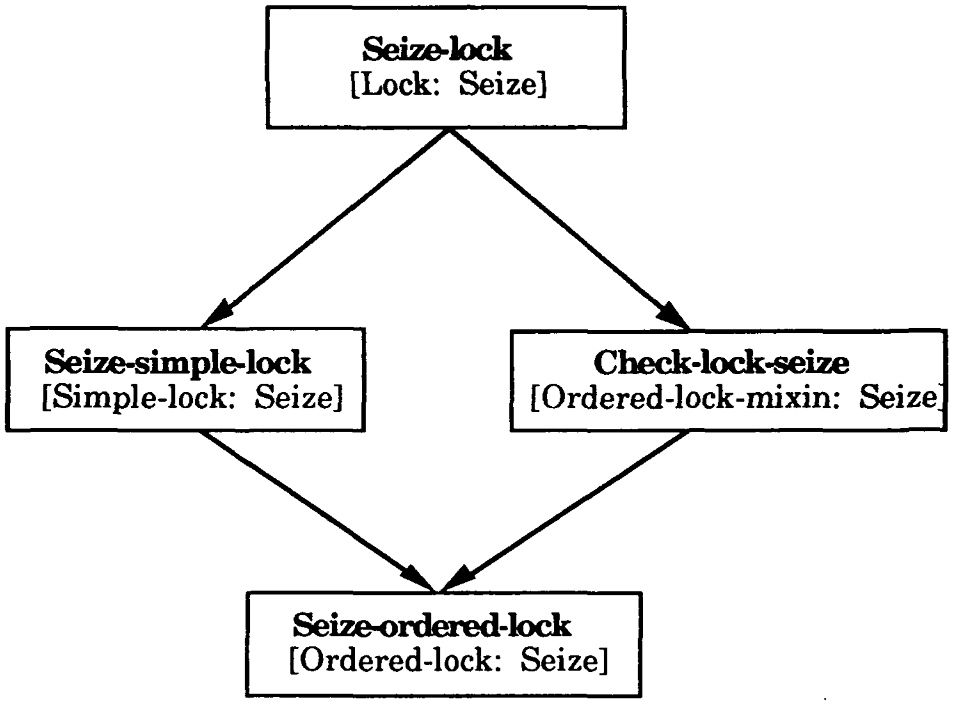


Figure 4-41
Method Taxonomy Diagram

4.7.1 IDEF4 Instantiation Language Primitives

In IDEF4, an instance of a class is represented graphically as a round-cornered box that is divided into two regions (see Figure 4-42). The field in the bottom-most region is used to uniquely identify the instance. (An object has state, behavior, and a unique identity.) The unique identifier is created by concatenating the class name of the instance with the number of the instance and enclosing them in angle brackets. For example, the thirteenth instantiation of the class Vertex has <*Vertex 13*> as its unique identifier.

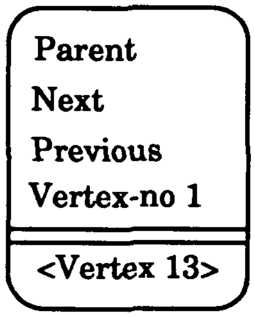


Figure 4-42
IDEF4 Instance of the Vertex Class

The instance attributes (value-returning, possibly computation-initiating) are listed in a column in the upper region of the IDEF4 instance box, just as they are in the IDEF4 class box. The IDEF4 instantiation language provides two ways to indicate the value assigned to or returned by the attribute. This is done by using value links or typing the unique identifier of the value instance to the right of the attribute. In the case of instances of numeric types, it is acceptable to type the value. This allows one to use the value 1 instead of <*Integer 1*> as a unique identifier for the integer 1. The IDEF4 value links are similar to the IDEF4 type links in that they start inside the IDEF4 instance box, next to thc attribute whose value is being annotated, and end in an arrow pointing to the boundary of an instance box as shown in Figure 4-44. The value link may be joined by several other value links that are pointing to the same instance, as is the case of the link from the parent attribute of the vertex instances depicted in Figure 4-44.

Figures 4-43 and 64 show a type diagram and instantiation diagram, respectively, for a triangle with vertices. The vertices have the necessary attributes for being part of a doubly linked list of vertices and an attribute for pointing to a parent -- which in this case is an instance of *Triangle*. The *Triangle* class has a *Vertex-ring* attribute that points to any instance on the doubly linked list of vertices. The *Vertex* and *Triangle* class have a number attribute whose type has been left unspecified in the type diagram as shown in Figure 4-43.

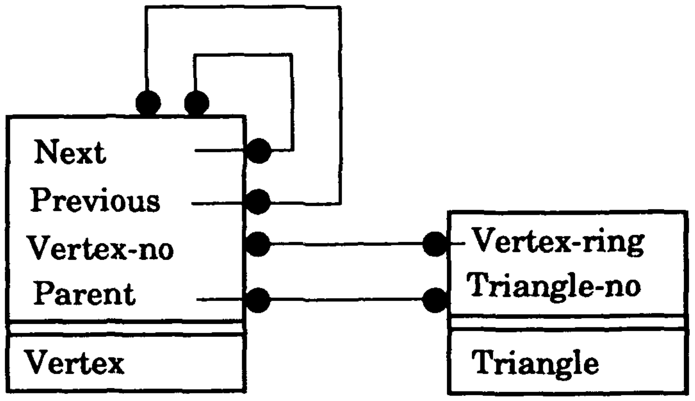


Figure 4-43
Type Diagram for Triangle Instantiation Scenario

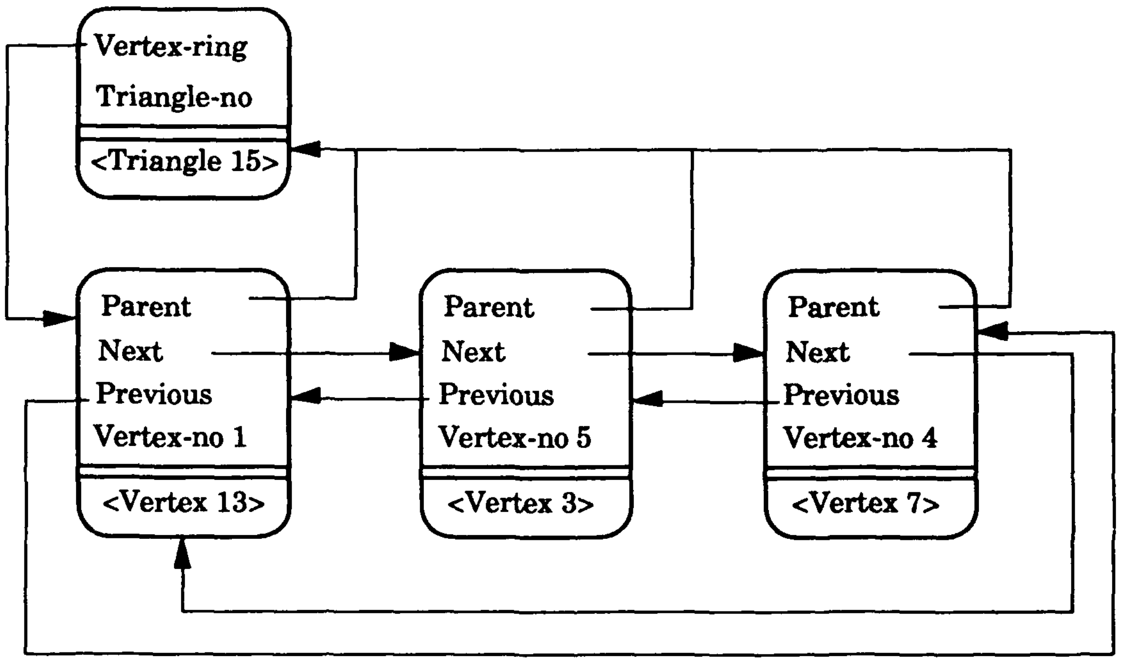


Figure 4-44
Instantiation of Triangle with Vertices

4.7.2 IDEF4 Design Validation

If we were to do an object-oriented design for a car, we would certainly have classes *Engine* and *Body*. The *Engine* class having an attribute *My-body* of type *Body* and the *Body* having an attribute *My-engine* of type *Engine* is shown in Figure 4-45. This design can easily and unambiguously be represented in IDEF4 by the type diagram shown in Figure 4-45.

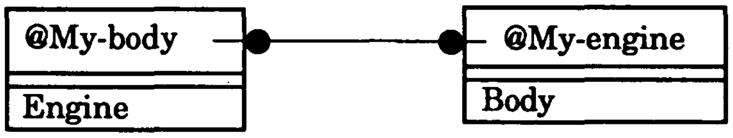


Figure 4-45
Type Diagram for Engine and Body

If we were to specify an object design for the three vertices of a triangle linked together in a circular, doubly linked list, it may not be so easy. The type diagram for this design is shown in Figure 4-46. The problem is remembering to define the constraints that specify that the previous vertex of the first is the third, and the next vertex of the third vertex is the first vertex. Specification problems can be discovered by designing primitive instantiation scenarios for the vertices (see Figure 4-47). These scenarios

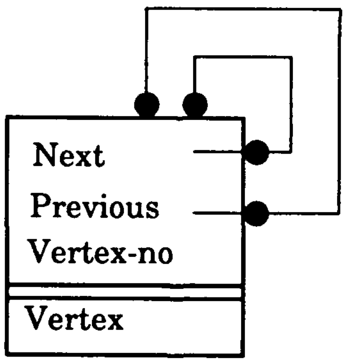


Figure 4-46
Type Diagram for Triangle Vertices

can be used to test whether the evolving design specification conveys what we had intended. In Figure 4-47, the states (i.e., the slot values of <Vertex 1> and <Vertex 4>) are identical; thus, if the constraints on the configuration of the circular, doubly linked list only referenced the attribute values (i.e., the state of the *Vertex* instances) and there was no constraint on the uniqueness of the *Vertex*-no attribute, then the system would be unable to distinguish between <Vertex 1> and <Vertex 4>. Additionally, a situation could arise for which the values pointed to by the *Next* and *Previous* slots of the vertices are consistent with the specification, but are used in a way that is inconsistent with the intended design specification (see Figure 4-47).

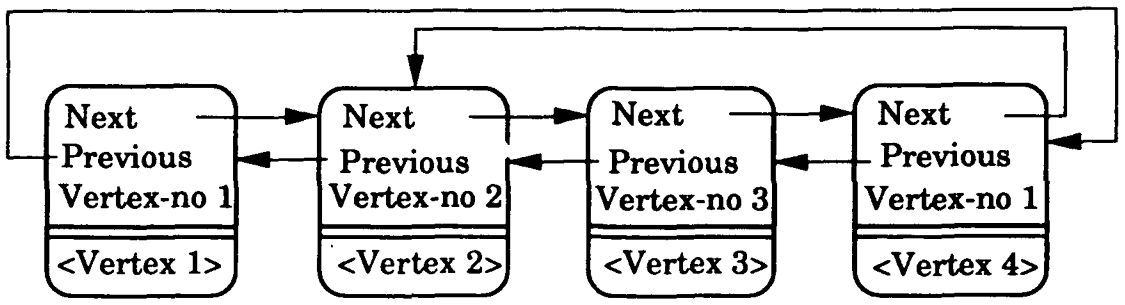


Figure 4-47
Erroneous Instantiation, Consistent with Design

5.0 IDEF4 Design Development Procedure

The design process is the predominant activity early in the software system development life-cycle. The design artifacts resulting from the design activity are important to the implementation and the subsequent sustaining activities. The notion of a design life-cycle is a convenient device often used to help produce an understanding of the basic design processes, particularly for administrative purposes. The design process from such a view is assumed to begin at some point, to continue through maturity, and eventually to stop. This view of design (as a series of incremental and sequentially interdependent steps) is an attempt to order the steps of the process such that each step can be looked on as an independent state, except for its occurrence relative to the other states that surround it. Use of a system often reveals problems that either are not addressed by the system or are products of changes in the system environment.

Design strategies can be considered "meta-plans" for dealing with the complexities of frequently occurring design situations. They can be viewed as methodizations or organizations of primitive design activities identified above. Three types of design strategies that can be considered.

1. External-constraint-driven Design - Design for situations in which the software goals. intentions, and requirements are not well-characterized, much less defined. These situations often result when the designer is brought into the product development process too early.

2. Characteristic-driven Design - Software design in a closely controlled situation for which strict accountability and proof of adequacy are rigidly enforced. These design situations often involve potentially life-threatening situations.

3. Carry-over-driven Design (Routine Design) - Changes to existing, implemented designs or designs that are well understood (e.g., sorting).

From an OOD point of view, the external-constraint-driven and carry-over-driven strategies are the most common design situations. The design development procedure, outlined in the following sections, is a distillation of the experience and insights gained in building several IDEF4 designs for different types of case studies.

5.1 Object-oriented Decomposition

IDEF4, as an OOD method, focuses on the description of the objects and types of behavior required of a system. The focus on the types of behavior (method sets) to be exhibited by the classes provides an appropriate means of partitioning systems into small, easily understood pieces. These types of behavior, paired with data types (as is done in OOD), provides for a more modular design; this results in implementations that have the desirable life-cycle characteristics for which object-oriented implementations are known.

Figure 5-1 illustrates that partitioning a system according to classes and their behavior provides modular composability by allowing functionality to be associated with the data. Thus, the design focuses on defining the types of behavior that the data types (classes) must exhibit.

Figure 5-2 illustrates the mapping between an IDEF4 Design and possible implementations. Each IDEF4 routine-class pair selects an IDEF4 method set. The IDEF4 class maps to an implemented class, and the IDEF4 routine maps to generic methods or messages in the implementation. Each class/generic method pair in the implementation selects a method in the implementation. This method must satisfy the contract on the method set selected by the IDEF4 routine-class pair associated with the implemented class and generic method, respectively. Thus, the contracts on the method sets and the class-invariant constraints on the IDEF4 classes place requirements on the implementation. The more restrictive these requirements, the fewer programming decisions the implementor will have to make, and consequently, the fewer possible implementations there will be.

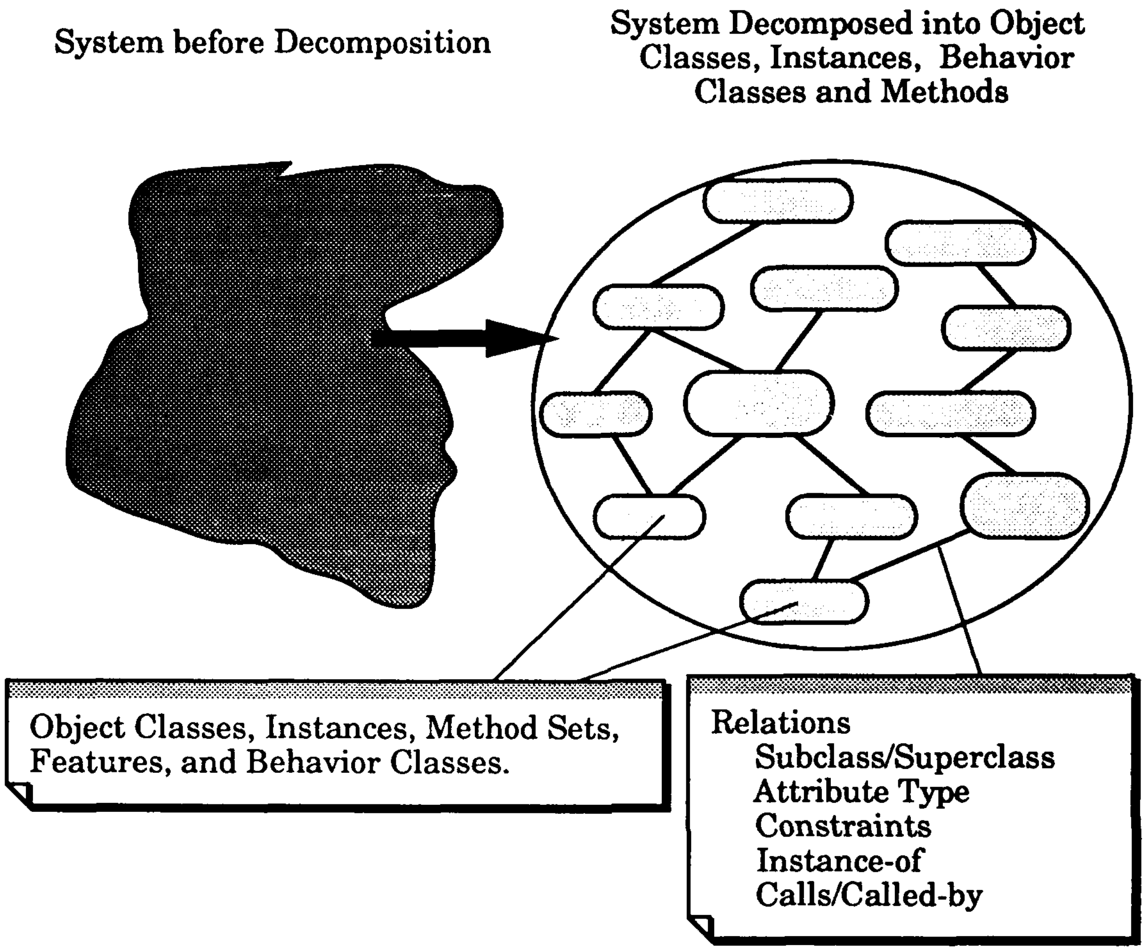

System before Decomposition
System Decomposed into Object Classes, Instances, Behavior Classes and Methods
Object Classes, Instances, Method Sets, Features, and Behavior Classes.
Relations
Subclass/Superclass
Attribute Type
Constraints
Instance-of
Calls/Called-by

Figure 5-1
Object-oriented Decomposition

5.2 IDEF4 Design Development Activities

After the IDEF4 method has been implemented, the following activities are used throughout the software design process.

1. Analyze evolving system requirements.
2. Develop class hierarchy (Inheritance Diagram).
3. Develop method taxonomy (Method Taxonomy Diagram).
4 Develop class composition structure (Type Diagram).

5. Develop protocols (Protocol Diagram).
6. Develop algorithmic decomposition (Client Diagram).

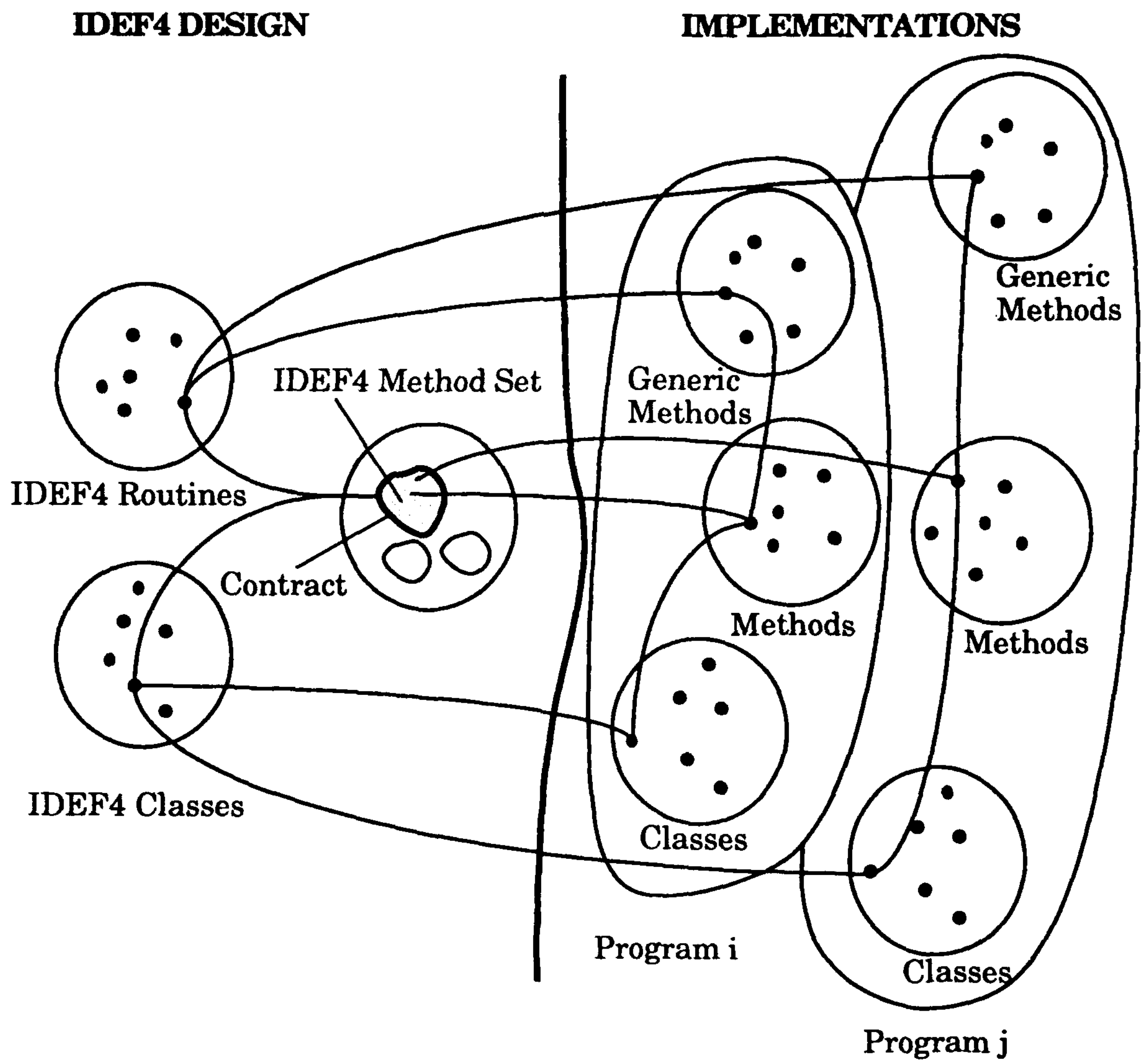


Figure 5-2
Implementation of IDEF4 Design

The dependency relations between these generic activities that have been discovered through the practice of IDEF4 are illustrated in Figure 5-3. Method sets and classes most often follow directly from the requirements, whereas the client diagrams are derived from the method sets and method contracts. The following sections will explain each of these activities in detail.

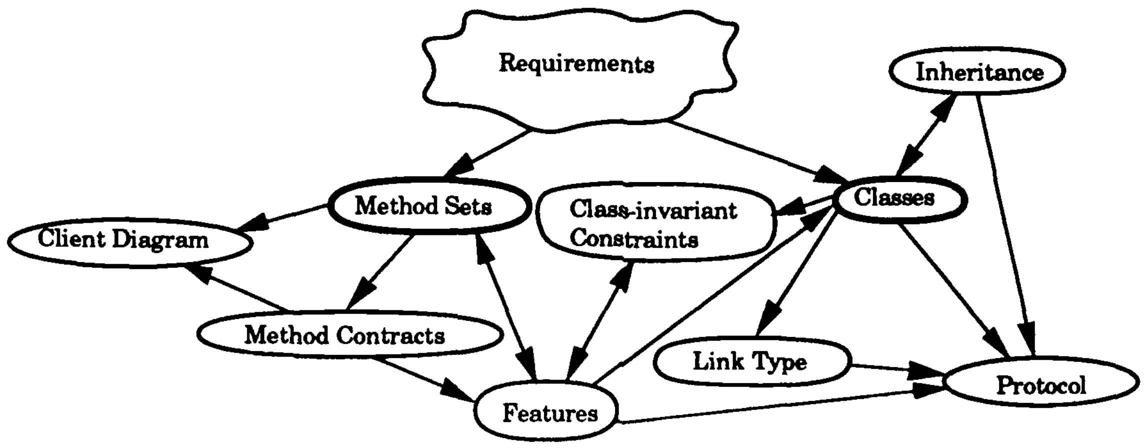


Figure 5-3
Causal Relations Between IDEF4 Elements

5.2.1 Analyze Evolving System Requirements

The first step system design development *is the analysis of the system* requirements. What will the system do and with what types of objects? This analysis involves more than just the system requirements. If functional, information, or process models are available, they can also be used to provide input at this stage. The purpose of this early analysis is to identify the intuitive classes and types of system behavior. In other words, the designer is partitioning the system into sets of coarse-grained object types, object compositions, and behaviors.

5.2.1.1 Identifying Initial Classes and Method Sets

In OOD, there is a tension between class decomposition, object composition, algorithmic decomposition, and polymorphic decomposition. There has been a void in current object design methods for supporting design trade-offs among object, class, method, and algorithmic structure. IDEF4 not only provides support for least-commitment modeling of each of these four perspectives of an object design, but allows one to compare and contrast the effects of design decisions in one perspective against the others. The least commitment philosophy, used in IDEF4, allows one to refine the design progressively, beginning with a coarse-grained initial design that satisfies the requirements, and then using the existing design iteration with the additional requirements as the requirements for the next design cycle, and so on until the design reaches maturity. Each design iteration reduces the number of possible implementations that would be consistent with the design (design correct). If the design iterations are taken to their logical conclusion, the design would have only one possible implementation (i.e., the design would be the implementation).

The requirements analysis should produce a list of initial classes and method sets. For those unfamiliar with the domain of the target system, the system requirements would be the major source for the initial classes and method sets. These lists will undoubtedly be incomplete -- little more than a listing of potential classes devoid of class features and inheritance structure. The initial method sets will further specify the system and will be based primarily on the system requirements. If a functional (IDEFØ) model of the TO-BE system is available, it may provide insights into desired system behavior and, thus, the initial method sets.

The ease with which these initial lists of classes and method sets are produced will be depend on two factors:

- the expertise of the designer(s) in the domain of the target system, and
- the clarity and detail of the system requirements.

During this stage of development, the designer is primarily identifying user/customer expectations of system capabilities and operation. Later, the designer will determine class structures and design method sets that provide the expected system behavior.

5.2.1.2 Using Other IDEF Methods in Analysis

The primary source to aid the analyst in understanding customer expectations of the system is the system requirements. However, because the customer cannot always be expected to have full knowledge of the true nature of the problem involved, designers should not limit themselves to a requirements document alone. Quick and accurate identification of the initial classes and method sets is critical to both the successful completion of the design and a full understanding of the nature of the problem; therefore, it is often advantageous to look to other sources of information on the proposed system and its environment. These sources may include functional, process, and information models as well as existing OODs. The IDEF family includes methods for constructing each of these types of models.

For functional modeling, IDEFØ can be used to assist designers in identifying concepts and activities that should be addressed by the system. IDEF1 can be used by designers to develop an understanding of the information the organization uses, and, therefore, must be maintained by the system. IDEF3 provides for process flow and object state transition descriptions that will assist designers in organizing their concepts of the inner workings of 1) the existing system, 2) the proposed system, 3) user interaction with the system, and 4) the state changes objects to be manipulated by the system would undergo. Although these methods provide much useful information, they are expensive and time consuming.

First, we examine functional modeling (in particular IDEFØ) as an aid in the IDEF4 design process. IDEFØ is a method for modeling functions and their associated concepts. However, since the primary emphasis in IDEFØ modeling is functional modeling, there may be serious consequences in IDEF4 OOD. Although the functions that the projected system must perform are important, IDEF4 is object-centered design and if too much emphasis is placed on the system functions, the resulting system design may be too functionally oriented rather than object-oriented. A system organized around functional decomposition tends to consist of tightly coupled modules that are difficult and costly to maintain, thus eliminating a major advantage of the object-oriented paradigm. Minor changes can mean major system rewrites requiring months of redesign and programming. Even with these drawbacks, we do not discourage the use of IDEFØ as a useful tool to assist the designer of an object-oriented system, but rather encourage its careful use. An IDEFØ model developed with an object-centered view provides valuable insight into initial classes and routines that the system requires. Finally, IDEFØ models are developed along the lines of functional decomposition. IDEFØ models that are developed to just two or three levels of decomposition are the most useful. Models with more detail tend to emphasize functional decomposition over object-oriented decomposition.

The IDEF1 method for information modeling and IDEF1X for data modeling provide input in the area of initial class identification and relationships between classes. However, once again we emphasize that care must be taken in their application. Both methods tend to be relational in nature; as such, they do not fully provide for an object decomposition of the system. With each of these methods, complete and fully detailed models should not be necessary; in fact, they could actually be detrimental to the development of an object-oriented system.

Perhaps the most useful IDEF method for the object-oriented designer is IDEF3. IDEF3 provides the designer with a method for developing system process flow descriptions and object-state transition descriptions. Development and use of these descriptions from an object-centered viewpoint can prove very helpful in class definition and initial behavior identification.

As always, the interaction between the user and the system must be acceptable if the implemented system is to be used. Object-oriented system designers must consider the man-machine interface. From a software development perspective IDEF3 is very useful in designing these usage scenarios. If the object-center view is selected, the system behavior can be partitioned into the common functions the objects must exhibit. All objects in the system will have certain common operations performed on them. For example, consider the deletion of a system object. An object-centered IDEF3 would allow the description of delete functionality common to all system objects. Finally, we look at IDEF3 object state-transition networks. These will assist the designer in determining pre- and postconditions for methods, design of transactions, and state changes in the system objects.

The usefulness of these methods to object-oriented design can be ranked as follows: IDEF3, IDEFØ, and finally, trailing far behind, IDEF1 and IDEF1X. The following points should be noted by those using these methods.

1. The models or descriptions should be developed with an object-centered view.
2. For a small- to-medium-sized system, the methods would be used if available, but it is unlikely that all types of models would be developed unless the system is large.
3. Complete or fully developed models and descriptions are generally not needed and may even be misleading.

5.2.2 Develop Class Hierarchy

The development of class hierarchies involves specifying classes, features, inheritance lattices, and class-invariant constraints. The designers will first partition the required system behavior into intuitive classes, then systematically refine them.

Developing class hierarchies involves identifying classes, features, and the inheritance relations among classes. This is most often performed in a recursive series of steps which includes the following.

1. Partitioning classes into more specialized sets according to like, behavior, and state.
2. Classifying classes against existing inheritance diagrams or specifying them using features and class-invariant constraints.
3. Assembling these classes into existing inheritance diagrams (and type diagrams).
4. Rearranging inheritance lattices to simplify their structure, possibly resulting in repartitioning.

5.2.3 Develop Method Taxonomy

The designers will initially partition the required system behavior into intuitive method sets. For each initial method set, client diagrams should be developed as necessary. The development of method taxonomy diagrams requires identifying constraints on the methods and the processes that are to occur in the completed system when different actions take place. This is performed in a recursive series of steps which includes the following.

1. Partitioning method sets into more specialized sets.
2. Classifying method sets against existing taxonomies or specifying them via new contracts.
3. Assembling these method sets and contracts into existing method taxonomy diagrams.
4. Rearranging the method sets and contracts in the method taxonomy diagrams to refine their structure, possibly resulting in repartitioning.

To someone not involved in the process, these actions may seem to occur in a very uncontrolled manner. In the design of a small system, recognition of clearly defined process steps may be negligible. In the selection and refinement of the initial methods, the designer is depending on the initial constraints on the system (i.e., the requirements). As the design process continues, the evolving method set specifications become additional constraints on the design.

5.2.4 Develop Class Composition Structure (Type Diagram)

The type diagram is used to illustrate clustering of classes by the type links between attributes (value-returning) and classes. For instance, the type diagram can be used to describe that a car has an engine and four wheels, and the engine has eight pistons with each attached to a crankshaft. The type diagram may use input from information and data models such as IDEF1 and IDEF1X on link types. Most of the information about the type links will be in the contracts of method taxonomies and in the constraints in CIDSs attached to inheritance diagrams.

5.2.5 Develop Protocols (Protocol Diagrams)

Protocol diagrams specify the valid input argument types and return value types that must hold for a routine-class pair (method set). This information may be derived from the class-invariant constraints, attached to a class referencing the method set and CDSs, attached to the method set that is selected by the routine-class pair. For example, on a draw routine-class pair (method set), the input arguments must be a window object and a graphic object, and the window object must be open for an image to be created. This constraint may be formally defined as:

graphic(x)^window(y)^open(y)^Draw(x y)->create-image(z x).

It is trivial to deduce that the draw routine-class pair (method set) should accept a graphic object and a window as arguments.

5.2.6 Develop Algorithmic Decomposition (Client Diagrams)

The client diagram illustrates the algorithmic or functional decomposition of method sets. For example, the *Redraw* routine-class pair (method set) calls the *Erase* routine then the *Draw* routine. Note that it is necessary to specify the routine-class pair and thus, indirectly, the method set on the client; however, only the routine need be specified on the supplier side. How one recognizes an opportunity to apply algorithmic decomposition comes from the observation that for a method set, the internal control logic is being repeated for specification of other method sets in other method taxonomies. In the *Redraw* example, we may notice that the flow of control of the *Erase* method set is similar to that of the *Draw* method set. We could actually specify *Erase* by calling to *Draw*. This would mean that *Redraw* and *Erase* are generic method sets; that is, for each new class that needs *Redraw* capability, only a *Draw* method would have to be specified.

5.3 IDEF4 Design Evolution Process

Development of any IDEF4 design involves the creation of 1) method taxonomy diagrams, 2) inheritance diagrams, 3) type diagrams, 4) protocol diagrams, 5) client diagrams, 6) CIDSs, and 7) CDSs. The CDS associated with a method set in method taxonomy diagrams is, in fact, the characteristic function of that method set. In other words, the method must be defined when it is specified. The CIDS specifies behavior common to all the instances of each class; thus, it is important to specify these constraints early in the design of classes. Each component contributes to the final design of the system. With the design process, as with any recursive process, process termination criteria are important. It is not possible to give precise criteria for the completion of design activities, only that the rate of change of the design will decrease as the design reaches completion. This occurs because each new constraint specified in the CDSs and class-invariant constraints places more and more restrictive requirements on the design.

The development of an IDEF4 design is a process of specifying the structure and behavior of an object-oriented program or system. In the development of the IDEF4 design, the following four general steps are applied recursively.

1. Partition - Partition evolving requirements into smaller, more manageable pieces.
2. Classify/Specify - Classify against existing definitions or specify a new design object.
3. Assemble - Incorporate design objects into existing structures.
4. Rearrange - Rearrange existing structures to take advantage of newly introduced design objects. This may result in an adjustment in the design partitioning which would cause another iteration in the design recursion.

The term "recursive application" implies that the same process of specification is normally applied to each element of the partitioning of the five diagram types as well as in the overall design development activity. This recursion continues until the prototype classifications of the resulting elements can be clearly established.

5.3.1 Partition

In system requirements analysis, designers partition the system into intuitive classes and method sets. It is often useful at an early stage to check whether generic algorithmic decomposition is possible, in order to spot opportunities to reuse design structures. It must be stressed that object-oriented or data decompositions are more stable over time than functionally oriented decompositions; therefore, it is important that any functional decompositions performed are generic with respect to the data types that may be involved in the implementation of the algorithm. The generic functional decomposition of *Redraw* is described in Figure 5-4.

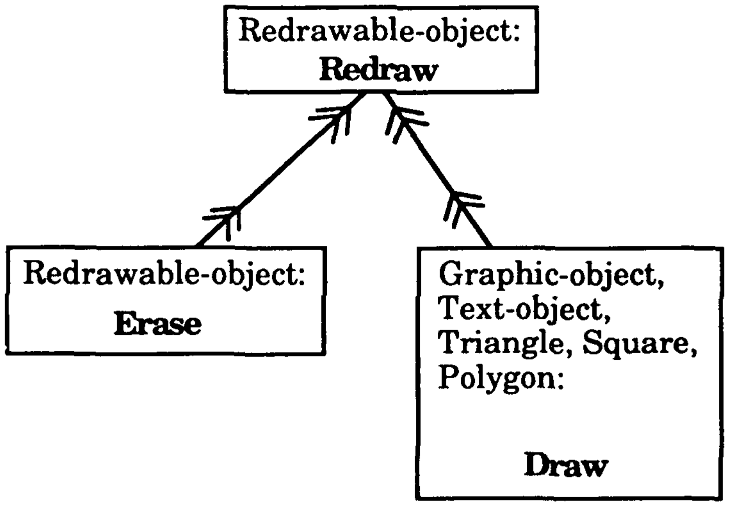

Redrawable-object: Redraw
Redrawable-object: Erase
Graphic-object, Text-object, Triangle, Square, Polygon: Draw

Figure 5-4
***Redraw* Client Diagram**

Generic functional (or algorithmic) partitioning of a design provides a means for decreasing the amount of code duplication when the system is implemented. In addition, it provides a vehicle for identifying portions of a design that can be reused or used to identify possible sections in which code from previous systems can be used in implementing the current design. For example, in Figure 5-4, *Redraw* was an initially identified type of behavior for a system. In the illustration, the link between the *Redraw* method set and the other two method sets shows a component-of relationship between the method sets *Erase* and *Draw*, and the method set

Redraw. Essentially, the designers are stating that the *Erase* and *Draw* method sets must be written such that they can be used by the *Redraw* method set. The purpose is to decrease the coding required by the *Redraw* methods (i.e., generalizing *Redraw)* and provide reusability of the *Erase* and *Draw* method sets. If the generic capability of *Redraw* was not acknowledged a *Redraw* method set would have to be designed for almost every class that used it. Essentially, the designers are stating in this illustration that *Redraw* has generic functionality and it will use the *Erase* and *Draw* method sets.

In the development of mixed diagram and the partitioning of the currently identified method sets, specification of the CDSs attached to each method set is extended. For example, in the discussion on Figure 5-4, the constraint on the *Redraw* method set was that it would provide part of its functionality via the *Erase* and *Draw* method sets. Furthermore, the *Erase* and *Draw* CDSs were also modified with the constraint that they be accessible and usable by the *Redraw* method set.

5.3.2 Classify/Specify

In the classification/specification phase, method sets and classes identified in the partitioning phase are matched against existing method sets and classes in order to classify them. If a method set or class cannot be classified, a specification must be designed. The specification of a class involves defining class-invariant constraints, protocol, and features; the specification of methods involves specifying CDSs and client diagrams.

As this generic activity is revisited recursively, specification of the classes and method sets will become more and more complete. Class and method specifications in each cycle may be viewed as additional requirements in the next design cycle.

5.3.3 Assemble

The classes and method sets that have been classified or specified must be assembled into the organizational structures employed in IDEF4. Newly classified/specified classes must be assembled into new or existing class-inheritance lattices (inheritance diagrams) and class-composition lattices (type diagrams). Newly classified/specified method sets must be assembled into new or existing method taxonomy diagrams and client diagrams. Mixed client/method taxonomy diagrams may be used to identify common behavioral or generic functionality. In the design of large systems, the assembly activity will likely occur in joint meetings between individuals responsible for different aspects of the design.

5.3.4 Rearrange

In the rearrange-design step, the designers will look for similarities in the structures, relating methods, or classes in order to identify opportunities for reuse of design structure. The rearrangement may necessitate alterations to the specifications of classes, methods, and their associated organizing structure. The designers may, for example, combine method sets and examine the CDSs associated with each to identify possible refinements. These refinements take the form of moving constraints up/down in the diagram without changing the behavior of either the super structure or the substructure or causing conflict with any evolving system requirements.

5.4 Organization of IDEF4 Design Documentation

A completed IDEF4 model will contain a class submodel and a method submodel. These submodels will be divided into five types of diagrams, two types of data sheets, and two types of dispatch mapping annotations. Depending on the size of the system being developed, there may be hundreds of pages of documentation. Navigating this documentation can prove to be an impossible task without some systematic cataloging of the diagrams, etc. In this section, we will describe the recommended organization of IDEF4 design documentation.

The object-oriented paradigm does not lend itself to sequential navigation of either a design or the associated program source code. Thus, the organization to the design document must provide for nonsequential access of the various design components. The navigation procedure should have some characteristics of hypertext documents; that is, from any component of the document it should be possible to get directly to any other component.

The basic organizing structure for the object-oriented paradigm is the class; therefore, IDEF4 documentation is centered on the class and routine names. The documentation is divided into two sections, the class submodel and the method submodel. Briefly, documentation is organized in the following framework.

1. Class Submodel

   - CIDS - CIDSs are the first components provided. These are ordered alphabetically by class name.

   - Inheritance Diagrams - The class inheritance diagrams are numbered sequentially in order of creation (e.g., I1, I2, . . ., In). Each inheritance diagram should have a unique title.

   - Dispatch Mappings on Inheritance Diagrams - Each inheritance diagram will be zero or one associated inheritance diagram that will show the dispatch mapping.

   - Type Diagrams - Type diagrams are placed after the inheritance diagrams in the class submodel and are numbered sequentially in order of creation (e.g., T1, T2, . . ., Tn). Each type diagram has a caption.

   - Instantiation Diagrams - Zero, one, or more associated instantiation diagrams will follow each type diagram.

   - Protocol Diagrams - Protocol Diagrams are placed in the class submodel after the type diagrams in alphabetical order by routine name (focus feature) followed by the class name (routine-class pair) with which they are associated.

2. Method Submodel

- CDSs - CDSs are ordered alphabetically by routine name followed by the class name (routine-class pair) with which they are associated. The method-set name will appear on the sheet following the routine-class pair name.

- Method Taxonomy Diagrams - The method taxonomy diagrams are listed in alphabetical order by the system routine for which they are named.

- Dispatch Mappings on Method Taxonomy diagrams - The associated method taxonomy diagram that shows the dispatch mapping will follow each method taxonomy diagram.

- Client Diagrams - Client diagrams are listed in alphabetical order by routine name (focus feature) followed by the class name (routine-class pair) with which they are associated.

Using a class name, a routine name, or a routine and class name combination as keys, any design component can be found directly -- with the exception of the inheritance and type diagrams which must be referenced via a CIDS. As seen in Figure 5-5, IDEF4 design documentation is organized around the CIDS which are alphabetized by class name. Each CIDS contains the numerical IDs of the inheritance diagram(s) and type diagram(s) in which the class appears. The inheritance and type diagrams are arranged numerically in order of creation. Each dispatch mapping inheritance diagram is placed in the documentation behind the diagram which it augments. The other components in the design are organized alphabetically by the routine or routine-class pair with which they are associated. Since the CIDS contains the list of all directly present features in the class, any component in the design related to a particular class can be easily located either directly in the case of the inheritance and type diagrams or indirectly via the class routines.

In practice, the likely starting point for any examination of the design will be the inheritance diagrams. The class and feature names that appear on these inheritance diagrams provide direct links to all of the other design components except the type diagrams, which can be reached by referring to the CIDSs.

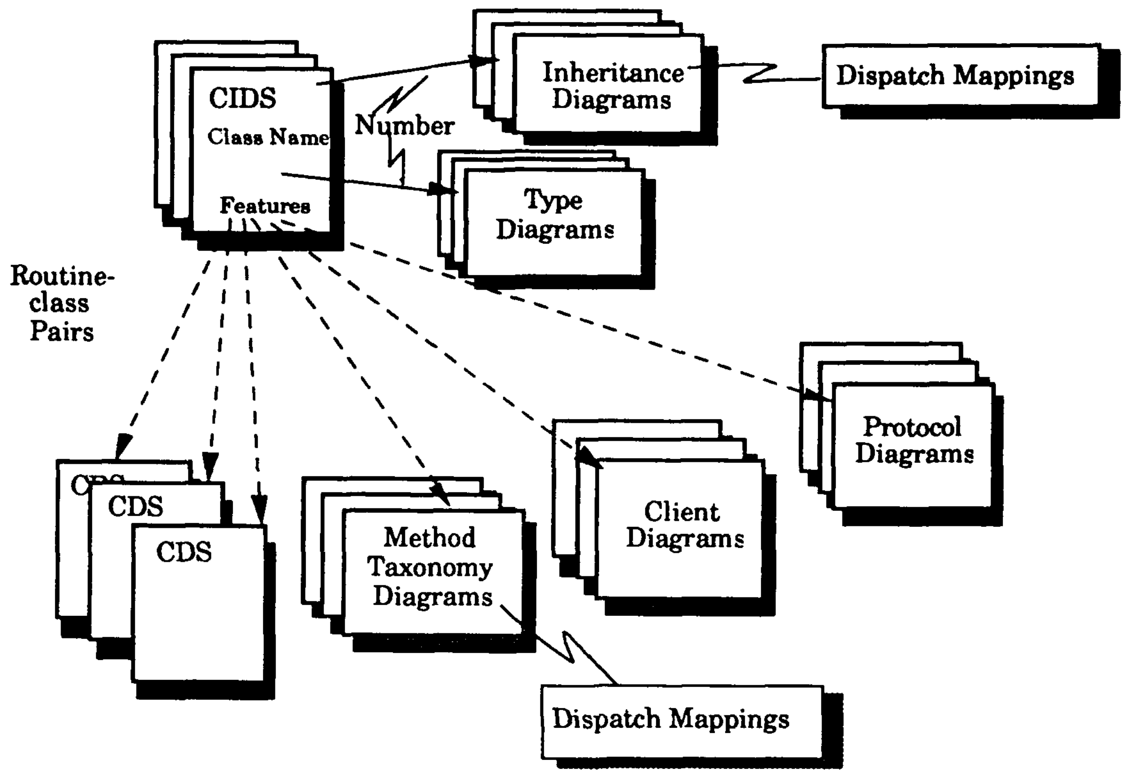


Figure 5-5
IDEF4 Document Organization

This organizational scheme allows the random perusal of a design, regardless of the starting point. For example, in Figure 5-6, the starting point is a method taxonomy diagram. Method taxonomy diagrams are alphabetized by the associated routine name. Given a specific method taxonomy in the documentation, the routine name is known (the name of the method taxonomy diagram). The CDSs, client diagrams, and protocol diagrams are associated with that routine are arranged alphabetically in the documentation first by routine name and second by class name. Using the routine name, one can easily assemble all the design components related to a particular routine. Any of these documents will provide pointers (class name) to the CIDS for the class of interest, which will provide access to the inheritance and type diagrams that the class appears in. Thus, it is possible to easily find the related components in a design.

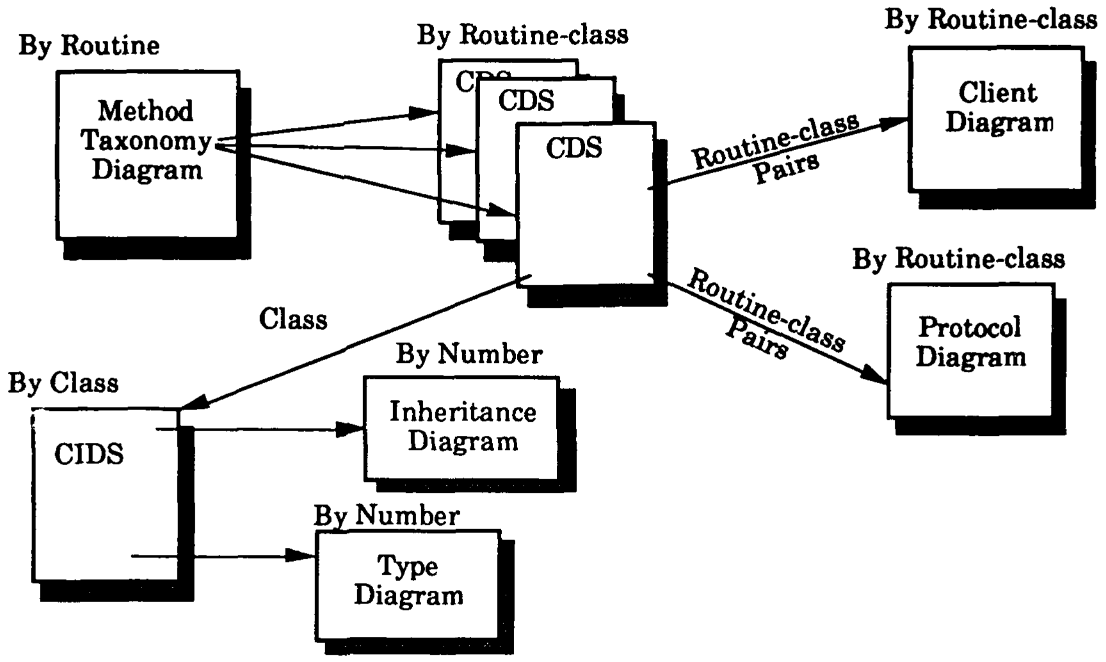


Figure 5-6
Navigating Documentation from the Method Taxonomy Diagrams

6.0 Tips and Traps in IDEF4 Design Development

The first design created by this method will be the most difficult. The designer and the person that implements the design will encounter many difficulties which will decrease with experience. This section will, hopefully, answer some of the questions the user will have when creating and interpreting early OODs. In this last section, we restate some basic characteristics of the IDEF4 object-oriented procedure and repeat some earlier statements about creating an IDEF4 design of an object-oriented system.

6.1 Fine-grained vs. Coarse-grained Methods and Classes

In the design of a system that is relatively small, initial classes may appear to be at a low level of abstraction and method sets may appear to be very fine-grained. Finer-grained detail in the initial method sets that are generated may also be caused by 1) the skills of the designer in the development of systems in this domain and 2) the clarity and simplicity of the system requirements.

Another factor contributing to the production of finer-grained methods and classes in the early stages of the design is in problem analysis. Designers who have functional (IDEFØ) and information (IDEF1) models of the proposed system will find that their initial listing of classes and methods will be much more complete and contain much more detail than those who do not have this input.

6.2 Method Taxonomy Diagrams and Routines

Method taxonomy diagrams provide an illustration of the relationships between the method-set contracts in the design. Each method set in the taxonomy is paired with one routine-class pair. As a rule, the routine paired with each method set in the taxonomy is the same; therefore, the convention is to name method taxonomy diagrams with the name of the system routine they describe (e.g., the *Sort* method taxonomy diagram or the *Seize* method taxonomy diagram).

Method taxonomy diagrams do not have to follow the class hierarchy. For example, when a feature in any class in the system implements a routine, the name of the feature is normally the name of the routine. However, this does not have to be the case. It is entirely possible and perfectly acceptable for the designer to specify two features in a class to perform the same routine. An example of this is the illustration described previously of a system in which the same class had two different sort routines defined. When this occurs, the primary routine for sort has the feature name *Sort* and the secondary routine was given some other name. The dispatching inheritance diagram shows that the features dispatch to different method sets but are in the same method taxonomy: *Sort*.

6.3 Routines and Routine class Pairs

Readers and users of IDEF4 tend to have some initial confusion regarding the terms routine, generic routine, and routine-class pairs. A routine refers to an IDEF4 feature that is computation initiating. A generic routine refers to a generic operation or function that the system or a hierarchy of classes is expected to provide. For example, the statement "The system is expected to provide for the sorting of files by name and number, sorting of lists of numbers, and sorting of lists of employee names" indicates that the system must provide for sorting different types of objects. Therefore, we say that the system will have a generic routine named *Sort*. When the meaning is clear from the context, a routine may be used in place of generic routine. A routine-class pair is an ordered pair that consists of the name of a generic routine and the name of a class. A routine-class pair is used as an index. If *Sort* is defined or redefined in the class *List,* the routine-class pair is called the *Sort:List* routine-class pair.

In IDEF4, a generic routine name is used to reference a method taxonomy diagram, which is a diagram of a particular type of system behavior. A routine-class pair references a method set (method-set contract) which is normally part of the method taxonomy diagram named for the generic routine. For example, the *Sort:List* routine-class pair is associated with an identically named method-set contract. That method-set contract will be represented by one of the nodes in the *Sort* method taxonomy diagram.

6.4 Multiple Return Types for One Return Value

It is not uncommon for a designer to want to show that a function will return a particular type for a successful execution and another type for an unsuccessful execution of the function. For example, the designer may want to indicate that the function *Fill-object-with-color* will return the color of the object. However, if the function cannot do this, it will return a string such as "No color possible." Neither the protocol diagrams nor the type diagrams of IDEF4 allows this type of information to be displayed. This information must be captured in either the CDS for the method set associated with *Fill-object-with-color* or by making the requirement part of the class-invariant where the feature is defined.

6.5 Characteristics of the Object-oriented Procedure

In the development of a design, designers begin with a coarse-grained system design, then expand and refine this until the design is complete. Experimentation with the IDEF4 design process has revealed several characteristics of the object-oriented procedure that are worthy of mention.

1. Specification of Requirements - Clearly stated system requirements will shorten development time by allowing the initial classes and methods to be more specific. These system requirements will be expanded and refined as the design process continues.

2. Understanding of Requirements - A definite relationship appears to exist between functional, information, process flow, and IDEF4 models as shown in Figure 5-7. For example:

   - The activities in a function model and the units of behavior (UOB) in a process description provide supporting evidence for method sets.

- The elaborations of a process flow description provide constraints for the CDS and CIDSs.

- The object state transition networks (OSTN) of a process description provide evidence of features, classes, and class-invariant constraints.

- The concepts (inputs, outputs, controls, and mechanisms) in a function model provide evidence for features.

- Information models provide evidence for object classes via entity classes, compositions via link classes, and features via attribute classes.

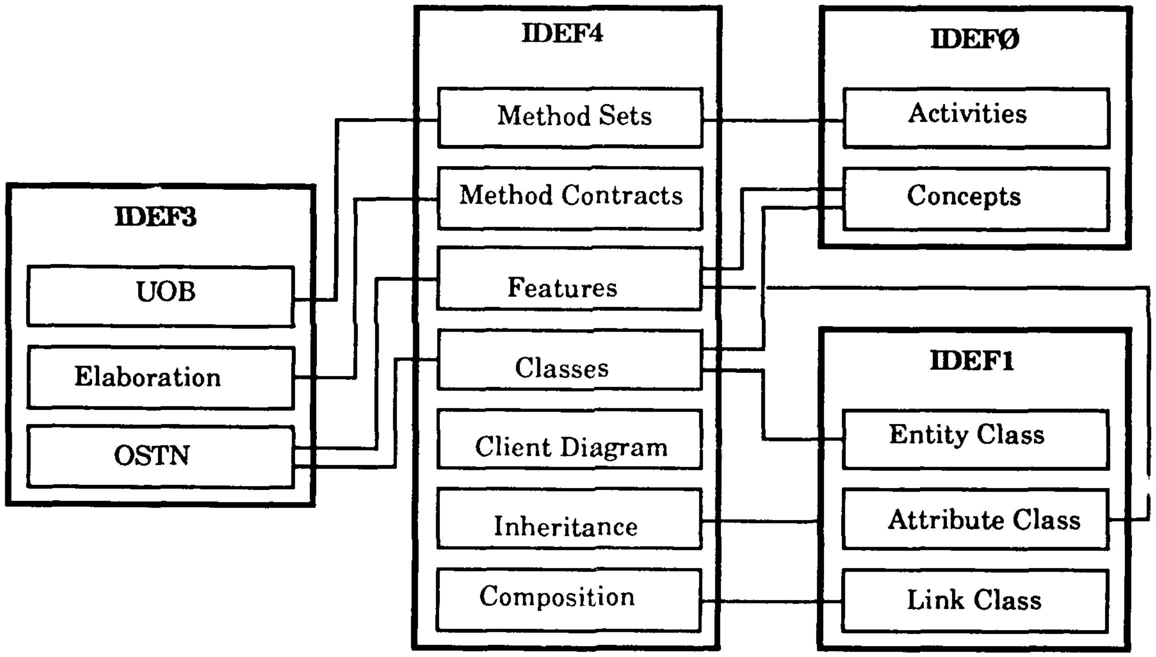


Figure 5-7

IDEF4 Relationship to IDEFØ, IDEF1, and IDEF3

3. Listings of Classes and Methods - Initial design phases should produce listings of classes and methods that are obvious from the requirements specification. No conscious attempt should be made in initial design phases to identify such things as attributes of the identified classes nor the algorithms by which the methods may execute.

4. Familiarity with System Type - If one or more of the participants in the design process are familiar with the type of system that is being designed, the design will likely produce more class substructure and more fine-grained method sets in the initial stages of design and will likely require less time to develop. If the system designers are not familiar with development of systems in the target system domain, the most notable difference will be longer design development time.

Reference List

Banerjee, J., Chou, H., Garza, J., & Kim, W. (1986). Data model issues for object-oriented applications. (MCC Technical Report. No DB-099-86). Austin, TX: Microelectronics and Computer Technology Corporation (MCC).

Bobrow, D. G., Kahn, K., Kiczales, G., Masinter, L, Stefik, M., & Zdybel. (1986). Merging LISP and object-oriented programming. In Proceedings of OOPSLA (pp. 17-29). Salem, MA: ACM.

Coleman, D. S. (1989). A Framework for Characterizing the Methods and Tools of an Integrated System Engineering Methodology (ISEM), Draft 2 Rev. 0. Santa Monica, CA. Pacific Information Management, Inc.

Gabriel, R. P., White, J. L., & Bobrow, D. G. (1991). CLOS : Integrating object-oriented and functional programming. Communications of the ACM, 34,(9). 29-38. Salem, MA: Communications of the ACM.

Goldberg, A. (1978). Smalltalk in the classroom. Palo Alto, CA: Xerox Palo Alto Research Center.

Itasca distributed object database management system [Technical Summary]. (1990). Minneapolis, MN: Itasca Systems, Inc.

KEE software development system user's manual. (1985). Menlo, CA: IntelliCorp.

Keene, S. E. (1989). Object-oriented programming in common LISP: A programmer's guide to CLOS. Cambridge, MA: Addison-Wesley.

Kim, W., Bertino, El., & Garza, J. (1988). Composite objects revisited (MCC Technical Report. No ACA-ST-387-88). Austin, TX: Microelectronics and Computer Technology Corporation (MCC).

Knowledge Based Systems Laboratory. (1991). IDEF4 technical report (KBSL-89-1004). College Station, TX: Department of Industrial Engineering, Texas A&M University.

Korson, T. D. and Vaishnava, V. K. (1986). An empirical study of the effects of modularity on program modifiability. E. Soloway & S. Iyengar, (Eds.), Empirical studies of programmers. Ablex Publishers.

Macintosh programmer's workshop PASCAL 3.0 reference. (1989). Cupertino, CA: Apple Computer.

MacLennan, B. J. (1982). Values and objects in programming languages. SIGPLAN notices, 17(12), 70-79

Mayer, R. J., et al., (1987). Knowledge-based integrated information systems development methodologies plan, (Vol. 2) (DTIC-A195851).

Mayer, R. J. (1991). Framework foundations research report [Final Report] . Wright-Patterson Air Force Base, OH: AFHRL/LRA

Mayer, R. J., Menzel, C. P., and deWitte, P. S D. (1991). IDEF3 technical report. WPAFB, OH: AL/HRGA.

Meyer, B. (1988). Object-oriented software construction. Santa Barbara, CA: Prentice-Hall.

Meyer, B. (1987). Reusability: The case for object -oriented design. IEEE Software, 4(2), 50-64.

Object store technical overview. (1990). Burlington MA: Object Design, Inc.

Orion Papers [Research Reports]. (1990). Minneapolis, MN: Itasca Systems, Inc.

Pascoe, G. A. (1986). Elements of object-oriented programming. Byte Magazine.

Soley, R. M. (1990). Object management architecture guide. Farmingham, MA: Object Management Group.

Stefik, M., and Bobrow, D. (1986). Object-oriented programming: Themes and Variations. The AI Magazine, 6(4), 40-62.

UNIX System V AT&T C++ language system, release 2.0 Product reference manual. (1989). Murray Hill, NJ: AT&T Bell Laboratories.

Yourdon, E., and Constantine, L. L. (1979). Structured design: Fundamentals of a discipline of computer program and systems design. Englewood Cliffs, NJ: Prentice-Hall.

Zachman, J. (1987). A framework for information systems architecture, IBM Systems Journal, 26(3), 276-292.

IDEF4 Glossary

Class	In object-oriented development, attributes and routines are identified with classes which provide the basic mechanism for encapsulation. The term "class" in the object-oriented world refers conceptually to certain data types with their associated operations which describe the features and behavior of a category of run-time computer program objects.
Class Submodel	The IDEF4 Class Submodel contains Inheritance Diagrams, Protocol Diagrams, and Type Diagrams.
Client Diagram	An algorithmic decomposition of a method set that illustrates "clients" and "suppliers" of method sets.
Contract	A method set is specified by the contract it fulfills. The contract is a declarative statement of the intended effect of the methods in the method set.
Client Diagram	Illustrates the algorithmic or functional decomposition of a method set by depicting "suppliers" and "clients" of method sets.
Contract Data Sheet	Specifies the contracts that methods in a method set must satisfy.
Design Method	A heuristic guide to thinking that applies a certain philosophy of design (Design Methodology).
Design Methodology	A philosophy of design (see also Design Method).
Dispatch Mapping	In IDEF4, the dispatch mapping connects the Class and Method Submodels. It is expressed by annotations on the Inheritance Diagrams and the Method Taxonomy Diagrams.
Functional Method	Returns a value and has no side effects.

Functional Method Set	Specifies methods that return a value and have no side effects.
Inheritance Diagram	Illustrates the subclass/superclass relation between classes.
Method	An implementation satisfying the constraints specified in a method set; uniquely identified by a class/generic-function pair.
Method Set	A set defined by a characteristic function called a method contract, defined in a contract data sheet. An IDEF4 method set is uniquely identified by an IDEF4 class-routine pair.
Method Submodel	The IDEF4 method submodel focuses on the structure of methods and contains Method Taxonomy Diagrams and Client Diagrams.
Method Taxonomy Diagram	Classifies method types by behavior similarity and links between class features and method types.
Procedural Method	Returns no value and has side effects.
Procedural Method Set	Specifies methods that return no value and have side effects.
Protocol Diagram	Specifies the protocol for method invocation.
Routine	A computation-initiating feature that may return values and have side effects.
Type Diagram	Illustrates composite relations among classes.

Index

☆U.S. Government Printing Office: 1992 — 648-069/40250